THE PARAGRAPH: DESIGN AND IMPLEMENTATION OF THE STAPL PARALLEL TASK GRAPH

A Dissertation

by

NATHAN LEE THOMAS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2012

Major Subject: Computer Science

THE PARAGRAPH: DESIGN AND IMPLEMENTATION OF THE STAPL PARALLEL TASK GRAPH

A Dissertation

by

NATHAN LEE THOMAS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Nancy M. Amato
	Jaakko Järvi
	Weping Shi
Head of Department,	Duncan M. Walker

May 2012

Major Subject: Computer Science

ABSTRACT

The Paragraph: Design and Implementation of
the STAPL Parallel Task Graph. (May 2012)
Nathan Lee Thomas, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. Languages and tools available for the development of parallel applications are often difficult to learn and use. The Standard Template Adaptive Parallel Library (STAPL) is being developed to help programmers address these difficulties.

STAPL is a parallel C++ library with functionality similar to STL, the ISO adopted C++ Standard Template Library. STAPL provides a collection of parallel `pContainers` for data storage and `pViews` that provide uniform data access operations by abstracting away the details of the `pContainer` data distribution. Generic `pAlgorithms` are written in terms of `PARAGRAPHs`, high level task graphs expressed as a composition of common parallel patterns. These task graphs define a set of operations on `pViews` as well as any ordering (i.e., dependences) on these operations that must be enforced by STAPL for a valid execution. The subject of this dissertation is the `PARAGRAPH Executor`, a framework that manages the runtime instantiation and execution of STAPL `PARAGRAPHs`.

We address several challenges present when using a task graph program representation and discuss a novel approach to dependence specification which allows task graph creation and execution to proceed concurrently. This overlapping increases scalability and reduces the resources required by the `PARAGRAPH Executor`. We also

describe the interface for task specification as well as optimizations that address issues such as data locality. We evaluate the performance of the `PARAGRAPH Executor` on several parallel machines including massively parallel Cray XT4 and Cray XE6 systems and an IBM Power5 cluster. Using tests including generic parallel algorithms, kernels from the NAS NPB suite, and a nuclear particle transport application written in STAPL, we demonstrate that the `PARAGRAPH Executor` enables STAPL to exhibit good scalability on more than 10^4 processors.

To my best friend Shawna.

There is no one else I would rather walk through the journey of life with.

To my children.

Reach for the stars.

ACKNOWLEDGMENTS

I am extremely grateful to my thesis advisor, Dr. Lawrence Rauchwerger, for his guidance and support during my graduate studies. After taking his compiler class, I became interested in systems research, eventually deciding to pursue a doctorate in the field. Over the years, we have had countless discussions about both research and life; I have greatly benefitted from his insights and experience. Most importantly, however, he demonstrated the resolve and consistency necessary to make steady progress in research.

Another member of my committee, Dr. Nancy M. Amato, co-directs the STAPL research project. She spent considerable amounts of her time reviewing software designs and suggesting alternative approaches when problems arose. Her constant concern with the STAPL user's experience challenged me to make things as simple and straightforward as possible.

I would also like to thank the other members of my committee. Dr. Jaakko Järvi provided useful feedback on both my research proposal and drafts of the dissertation. He also served as my mentor for the Graduate Teaching Academy program I participated in through the university. Dr. Weiping Shi also provided feedback and was very accommodating in scheduling time to serve on the committee. I am sincerely appreciative of both of their time and effort.

I would also like to thank Dr. Mauro Bianco, who was a postdoctoral researcher in our group for some time. We had many fruitful discussions that helped focus my thinking about the research problems I was investigating. He also reviewed design proposals and helped refine the interfaces of various software modules.

Dr. Timmie Smith and Dr. Gabriel Tanase joined the STAPL group as students around same time as me, and we spent much time together during graduate school.

As colleagues, we endlessly debated research problems and design choices while developing STAPL. As friends, we consoled each other when needed, in addition to celebrating successes. I respect them both immensely and value the impact they had on me.

Being part of the Parasol Laboratory, I was fortunate to interact with a relatively large number of graduate students working in similar areas. Many have contributed to my education and research in one way or another, including aiding in the development of STAPL, employing STAPL in their applications, or just through informal discussions we had about each other's research. These students include Alin Jula, Ping An, Silvius Rus, Chidambareswaran Raman, Tao Huang, Steven Saunders, William McLendon, Antal Buss, Adam Fidel, Ioannis Papadopoulos, Shuai Ye, Harshvardhan, Mani Zandifar, Jeremy Vu, Olga Pearce, Tarun Jain, Shishir Sharma, Xiabing Xu, Sam Jacobs, and Roger Pearce.

My wife, Shawna, has been simply amazing during these years. Her consistency and great attitude while we were in graduate school kept me moving forward. With this season of life closing, I am excited to discover the adventures and challenges that lie ahead for us. In the last couple of years, our children Daniel and Samantha have entered our world and turned it upside down. I thank them both for reminding me what true joy is all about.

Finally, I have been blessed with incredible parents. Although I only knew my late father for twelve years, he left an indelible mark on my life that helps guide me to this day. My mother, Belinda, has always been supportive, buying me my first computer and wisely transitioning me to a high school that could encourage my technical interests. Thanks mom, for everything.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. The Standard Template Adaptive Parallel Library (STAPL) [4, 5, 10–12, 52–54, 58] is being developed to help programmers address the difficulties of parallel programming. STAPL is a parallel C++ library with functionality similar, but not limited to that of STL, the ISO adopted C++ Standard Template Library [42]. STL is a collection of containers, iterators, and basic algorithms that can be used as high-level building blocks for sequential applications. Analogous to STL, STAPL provides a collection of parallel and distributed `pContainers` and `Views` that provide uniform data access operations by abstracting away the details of the particular `pContainer` being used to store data. In addition to parallel equivalents of STL algorithms, STAPL provides a framework to develop new generic `pAlgorithms` using `PARAGRAPHs`, high level task graphs written as a composition of common parallel patterns.

While STAPL shares some features with previous efforts to simplify parallel programming, there are several design decisions that uniquely position it. Previous generic parallel programming efforts, such as PSTL [34], aimed to maintain STL compatibility. In contrast, STAPL borrows heavily from the STL philosophy, but has chosen to refine some of its interfaces to better suit programming for large scale parallel systems.

Partitioned global address space (PGAS) languages [13, 15, 20, 59] help users explicitly manage data and work distribution to maximize performance. While there

---

This dissertation follows the style of *IEEE Transactions on Automatic Control.*

are some parts of STAPL that are aware of an application's data distribution, such as the `pContainers`, STAPL provides a programming model that does not require application developers to manually manage locality. `pViews` (Section III.2) as well as several techniques we present in this dissertation (Chapter VI) help achieve this simplified programming interface.

Projects such as [9,15,20,26] have sought to efficiently exploit nested parallelism. STAPL also exploits nested parallelism, though it seeks to separate the algorithmic expression of this hierarchical parallelism from its mapping onto a specific target machine for a given program's input. This decoupling increases both portability and the library's ability to accommodate input sensitive applications.

Finally, STAPL has chosen to use a task graph based program specification, an approach used by several other research efforts [22, 33, 36]. Instead of viewing algorithms as a *sequence of computation steps* to transform input into output [19], these approaches see them as a *collection of tasks*. A *task* performs some step of the computation, performing work on an intermediate set of values to produce an output. When one task produces some intermediate value required as an input to another task, there exists a *dependence* between them. This dependence must be enforced, requiring an ordering between the tasks, to ensure the expected output is produced. Together these tasks and their dependences form a *task graph*.

The task graph representation is fundamental to describing parallelism; when the tasks are single instructions and scalar data, one has a program representation that expresses the fundamental limits of concurrency, independent of any given architecture. However, this exact, fine grain representation is typically impossible to maintain due to resource restrictions; the memory required to represent it offsets the high degree of parallelism exposed. Next, the initialization of the representation at runtime often exhibits poor scalability, reducing the performance of the parallel ap-

plication employing it. Furthermore, task graph representations often have difficulty mapping onto varying architectures with differing number of processing elements. In addition to these portability issues, task graphs often struggle with dynamic applications, those whose task graph structure is dependent on the input data or partial program results.

This dissertation describes the `PARAGRAPH Executor`, an infrastructure in STAPL that attempts to addresses some of these common problems with task graph based representations. As we will see, the STAPL programming model lends itself towards managing the granularity of the application's task graph. We also present techniques to increase the scalability of graph creation, by overlapping this activity with task graph execution. These approaches also prove useful to specify the task graphs of dynamic applications. Finally, we represent optimizations to the `PARAGRAPH Executor` that place tasks on a system's processors with knowledge of the input data's distribution.

I.1. Research Objective and Contributions

In order for the task graph approach to program specification to be practical for parallel application development, the representation must be properly mapped to a target system and capable of taking into account any runtime affects an application's input has on its structure. Translating the `PARAGRAPH`'s high level expression of the STAPL programmer's intent into an efficient execution is the intent of our research. The `PARAGRAPH Executor` infrastructure presented in this dissertation makes several contributions towards this end:

- **Support for dynamic, arbitrary task graphs.** Input sensitive applications often have portions of the task graph that cannot be fully specified prior to

execution. We describe a taxonomy of task types that allows both data and task level parallelism to be expressed in a unified manner. In order to maintain scalability, we allow the creation of these tasks to proceed asynchronously with regard to others, even those with which they share a dependence relationship. `PARAGRAPH` initialization and execution proceed concurrently.

- **Support for incremental task graph generation.** Storing the complete, runtime task graph of large applications can prove taxing on the system. The more resources that are delegated to this representation, the less are available to service the real computation. Additional incremental generation is needed for some dynamic programs, where the amount of work that must be done cannot be determined at compile time. To address this concern, we support incremental graph generation, execution, and reclamation of utilized resources.

- **Locality based optimizations.** The *placement* of tasks on processing elements for *execution* is independent of the element that *created* the task, and can be customized based on the computation and data to minimize latency. After task placement, the method used to *access* the data is optimized based on a dynamic inspection of the data distribution.

We utilize these contributions in both algorithms and full applications in the evaluation chapter of this dissertation. In our experiments, we look at parallel equivalents of common STL algorithms. We then consider a subset of the NAS parallel benchmarks [7]. We finally present a full nuclear particle transport equation written in STAPL that also makes use of the `PARAGRAPH Executor`. Additionally, there are many publications reporting on STAPL components [4, 5, 10–12, 52–54, 58] that use the `PARAGRAPH Executor` presented here to manage the execution of algorithms and

applications that they present. Publications that are more directly related to this dissertation are currently being prepared.

I.2. Outline

The remainder of this dissertation proceeds as follows. We begin with discussion of related work in Chapter II. In Chapter III, we provided an overview the STAPL parallel library, the context in which this research occurs. We primarily focus on the components that the `PARAGRAPH Executor` interacts with. The `PARAGRAPH` provides the specifications for task graphs that must be executed. The `pView` describes the data used by these tasks and gives the task graph locality information to aid in efficient execution. Finally, the runtime system (RTS) assists in task execution and abstracts communications through its remote method invocation facility (`ARMI`).

In Chapter IV, we discuss the interface of `PARAGRAPH Executor` beginning with a description of its general form and hierarchical nature. We then formalize the problem of specifying dependences between tasks in Section IV.2, describing traditional approaches as well as modifications we employ to enable overlapped task creation and execution. In Section IV.3 we then describe the taxonomy of *workfunctions* used in STAPL to define the computation that tasks perform.

We turn our focus toward the implementations of the `PARAGRAPH Executor` in Chapter V, describing the task creation and execution processes in Section V.3. These activities drive the event driven *edge container* (described in Section V.5), the component responsible for the maintenance and enforcement of task dependences as well as the propagation of a computation's intermediate values. We also discuss the runtime system's *scheduler* and the incremental reclamation of system resources that takes place during the execution of the task graph.

In Chapter VI, we outline several optimizations that are important for efficient executions. Employing a customizable policy, *task placement* allows tasks to be migrated from their locale of creation to another prior to execution, to address locality and load balancing concerns. After task placement, *view localization* allows us to optimize access to data, removing the overhead of the conservative approach initially used when the relation between task and data distributions cannot be statically determined.

We evaluate the performance of the `PARAGRAPH Executor` in Chapter VII, where it is used to execute algorithms and applications written in STAPL. We show experimental results for STL equivalent algorithms and a subset of the NAS [7] parallel benchmarks. We also look at a larger application employing STAPL, from the domain of nuclear engineering. We close with conclusions and discussion of future work in Chapter VIII.

CHAPTER II

RELATED WORK

Many compilers use some form of task graph for internal program representation. The Hierarchical Task Graph (HTG) [28] was implemented in the Parafrase-2 [45] compiler as an intermediate form that encoded the data and control dependences of an application in order to enable task parallelism. At each level of the graph hierarchy, a task is a sequence of instructions; and tasks at the same level of the graph may be executed in parallel if there is no dependence between them. HTGs allow nested task parallelism; independent tasks at any level of the graph can be executed in parallel. STAPL task graphs employ this same hierarchical approach and can be seen as a form of intermediate program representation, prior to mapping on the system for execution. However, instead of being a *static* artifact, *extracted* from the application's source, our task graph is dynamic, and based on a direct graph specification by the developer via a `PARAGRAPH`.

The Pegasus Workflow Management System [22] is a tool that coordinates the execution of large-scale scientific applications across multiple platforms in different locations. Pegasus handles the same operations as the STAPL *scheduler*, which we discuss in Section V.6. Workflows can be expressed using multiple tools that differ in their level of abstraction. Pegasus is capable of handling explicitly specified workflows, Chimera [24] specifications, and CA [35] specifications. The first two levels of specification correspond loosely to the explicit task creation and task factory specification discussed in Section IV.3. The third is similar to a composition of task graphs that the `PARAGRAPH` distills from the user. The Chimera system contains a collection of data sets that represent the results of scientific computations. A scientist can query the system and specify a new data set that are needed. Chimera uses its knowledge

of the results it already has and the computations to perform on them to generate a directed acyclic graph (DAG). The DAG can then be given to Pegasus or another workflow management system to schedule the necessary computations. Though operating at a much higher level than STAPL, the general idea of using task graphs to guide computation is the same.

Intel Concurrent Collections [14,36] allows a high level, dataflow-like description of an application. An application developer expresses the data to be processed in *item collections*, the computation to perform in *step collections*, and any control dependence between step collections in *tag collections*. Each of these concepts is referred to as a collection to emphasize that it is a set of distinct units. The step instances contained in a step collection are the minimal execution unit and schedulable entity, and item instances are the finest grain of communication that can occur in the application. The result of the developer's work is a model of the application that is fine-grained. This model is then passed to a *tuning expert*, responsible for mapping the computation onto the system for efficient execution. Note that this expert may be either software that automates the process or a human that applies the necessary transformations manually. The primary responsibility of the tuning expert is to group items together into coarse-grained entities to minimize runtime overhead, and to place them within a process for execution [41].

This decoupling of application specification and task preparation for execution is the same approach we take in STAPL. The `PARAGRAPH` created by the user expresses the problem at high level. The pattern library present in `PARAGRAPH` then collaborates with the `PARAGRAPH Executor` to create these tasks, using the task factories discussed in Section IV.3.6. It is then the responsibility of the `PARAGRAPH Executor` to place these tasks in system, enforce dependences, and enable access to intermediate data.

Intel Threading Building Blocks [33] (TBB) provides parallel primitives such as

`parallel_for` and `parallel_reduce`, implemented using a split / join task pattern similar to Cilk [26]. This task specification is dynamically specified either within the pattern, or it can also be employed directly by the user. A recent version of the library introduced a more general task graph class as a community preview feature. This class is instantiated and then explicitly populated with nodes and edges. The behavior of a node varies based on its type. There are nodes that execute user functions in addition to nodes that simplify setting up different communication patterns. After a graph is constructed it may be repeatedly executed. However, in contrast to our approach this specification must be complete prior to the start of executing the task graph.

Algorithmic skeletons [17, 18, 21] are higher-order functions that implement the structure of a parallel algorithm and accept functions that implement the operations to be performed within the algorithm as arguments. An example of a skeleton is *reduce*, which can be thought of as a parallel implementation of the STL accumulate algorithm. The algorithm accepts a function object that implements the operation to be applied on the elements to form the accumulation. How the code within the algorithm is parallelized and how the partial results are combined to form the final answer is hidden from the user. The DatTeL library [8] is a skeleton library for C++ that implements parallel versions of the STL algorithms as skeletons using this approach.

The task factories offered by the `PARAGRAPH` in collaboration with the `PARAGRAPH Executor` infrastructure are similar in form to skeletons. STAPL implements common skeleton patterns such *map, reduce, and prefix scan* as factories. `pAlgorithms` use these patterns to easily instantiate the `PARAGRAPHs` needed to perform the desired computation. While user code written using skeletons is portable, the skeleton implementation usually is not; skeletons are typically implemented using the native run-time system or a low level API such as MP [51], and the implementation of

each skeleton is independent of the others in a library. In contrast, both user code and the task factory based pattern implementations are portable in STAPL. The latter is achieved by dynamically selecting the granularity and placement of tasks in the system, employing abstracted locality information provided by the `pView`. Note that Intel Threading Building Blocks provides `parallel_for`, `parallel_reduce`, and `pipeline` based algorithms which are skeletons whose implementations also use a common, higher level of abstraction for their implementation.

Linda [27] and related tuplespace approaches [25, 39] are not manifestations of the task graph based programming paradigm, but it bears some relation to our work through its notion of a global, associative memory store. Users create entries in this store, dubbed the tuplespace, and subsequently create tasks to evaluate these entries. The result of this computation is written back to the tuplespace, where it can be subsequently read by other processes before being explicitly removed. Therefore, while there is a producer consumer pattern in use, the associated task graph is not expressed to and subsequently managed by the library, but instead is manually managed by the developer. The *edge container*, presented in Section V.5, also acts as a logically shared storage for intermediate values, indexed by the task that created it. However, we leverage knowledge available to us about task dependences, optimizing the dissemination of intermediate values and automatically handling the retirement of values that are no longer needed.

In a wider context, there is a relatively large body of work that has similar goals to STAPL. PSTL [34], POOMA [47], and STAPL borrow from the STL philosophy, i.e., they provide concepts such as containers, iterators, and algorithms. The Parallel Standard Template Library (PSTL) had similar goals to STAPL; it uses parallel iterators as a parallel equivalent to STL iterators and provides some parallel algorithms and containers. However, PSTL was focused on STL compatibility, while STAPL extends

STL by introducing additional parallel data structures and task graph specification of generic parallel algorithms.

Projects like NESL [9], CILK [26], Split-C [20], Chapel [13], and X10 [15] provide the ability to exploit nested parallelism. In addition to nested parallelism, STAPL is intended to automatically generate recursive parallelization without user intervention. Several languages/libraries abstract shared memory machines, hiding the details of the data distribution from the user. Others provide the user with a *partitioned global address space (PGAS)*, including Split-C [20], X10 [15], Chapel [13], and Titanium [59]. In PGAS languages, memory accesses have different costs for local and remote data. STAPL provides a shared memory abstraction for the naive user while exposing a PGAS architecture to the more advanced user, decoupling the application from library development.

CHAPTER III

THE STAPL DEVELOPMENT ENVIRONMENT

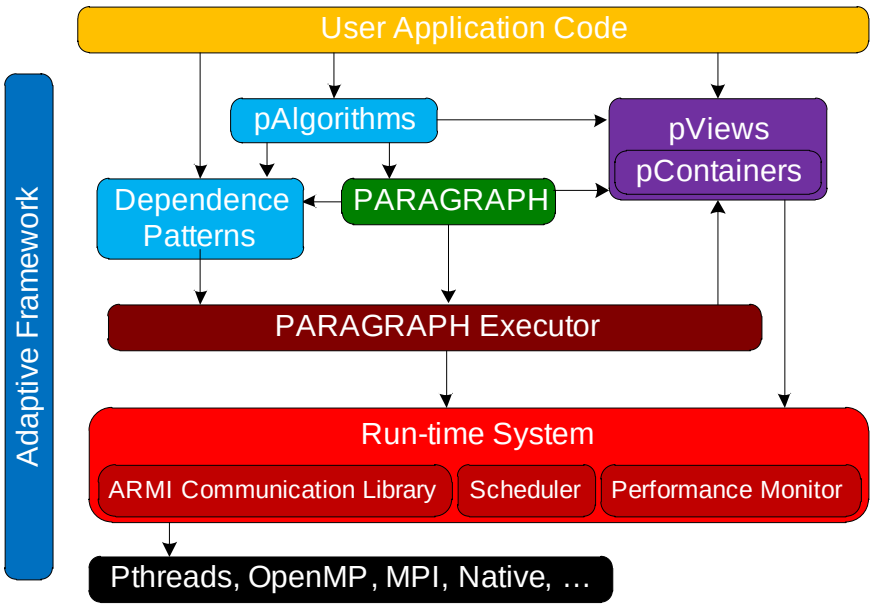


Fig. 1.: STAPL overview.

STAPL [11, 46, 48, 58] is a framework for parallel C++ code development whose major components are shown in Figure 1. Its core is a library of parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`) [52] that have interfaces similar to the (sequential) C++ standard library (STL) [42], as shown in Figure 2. Analogous to STL algorithms that use *iterators*, STAPL `pAlgorithms` are written in terms of `pViews` so that the same algorithm can operate on multiple `pContainers`.

`pAlgorithms` are a composition of computational patterns represented by `PARAGRAPHs`,

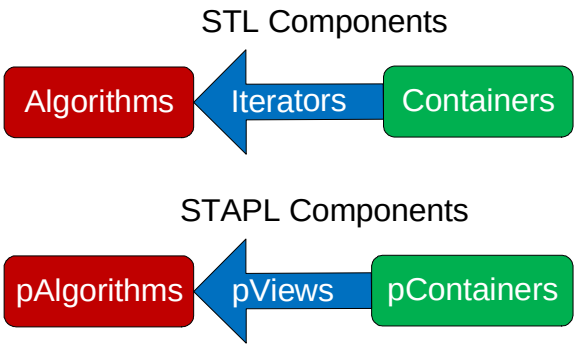


Fig. 2.: Parallels between STL and STAPL components.

which are graphs whose vertices are tasks and edges are dependences, if any exist, between tasks. A task includes both *work* (*workfunctions*) and *data* (from `pContainers`, generically accessed through `pViews`). The `PARAGRAPH Executor` is responsible for the runtime instantiation and execution of parallel computations represented by `PARAGRAPHs`. Nested parallelism can be created by invoking a `pAlgorithm` from within a task.

III.1. pContainers

STAPL `pContainers` are distributed, thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They are composable and extendible via inheritance. Currently, STAPL provides counterparts of all STL containers (e.g., `pArray`, `pVector`, `pList`, `pMap`, etc.), and two `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`). `pContainers` consist of a set of `bContainers`, that are the basic storage components for the elements, as well as distribution information that manages the distribution of the elements across the parallel machine.

`pContainers` provide methods corresponding to the STL container methods, and some additional methods specifically designed for parallel use. For example, STAPL provides an `insert_async` method that can return control to the caller before its execution completes, or an `insert_anywhere` that does not specify where an element is going to be inserted and is executed asynchronously. While a `pContainer`'s data may be distributed, `pContainers` offer the programmer a *shared object view*, i.e., they are shared data structures with a global address space. This is supported by assigning each `pContainer` element a unique global identifier (GID) and by providing each `pContainer` an internal translation mechanism that can locate, transparently, both local and remote elements. The physical distribution of `pContainer` data can be determined automatically by STAPL or it can be user-specified.

III.2. pViews

Decoupling of data structures and algorithms is a common practice in generic programming. STL, the C++ Standard Template Library, obtains this abstraction by using *iterators*, which provide a generic interface for algorithms to access data that is stored in containers. This mechanism enables the same algorithm to operate on multiple containers. In STL, different containers support various types of iterators that provide appropriate functionality for the data structure, and algorithms can specify which types of iterators they can use. For example, algorithms requiring write operations cannot work on input iterators and lists do not support random access iterators. The major capability provided by the iterator is a mechanism to traverse the data of a container.

The STAPL `pView` [10] generalizes the iterator concept by providing an abstract data type (ADT) for the data it represents. While an iterator corresponds to a single

element, a `pView` corresponds to a collection of elements. Also, while an iterator primarily provides a traversal mechanism, `pViews` provide a variety of operations as defined by the ADT. For example, all STAPL `pViews` support size() operations that provide the number of elements represented by the `pView`. A STAPL `pView` can provide operations that return new `pViews`. For example, a `pMatrix` supports access to rows, columns, and blocks of its elements through different `pViews`.

`pViews` have *reference semantics*, meaning that a `pView` does not own the actual elements of the collection but simply *references* to them. The collection is typically stored in a `pContainer` to which the `pView` refers; this allows a `pView` to be a relatively light weight object as compared to a container. However, the collection could also be another `pView`, or an arbitrary object that provides a container interface. With this flexibility, the user can define `pViews` over `pViews`, and also `pViews` that generate values dynamically, read them from a file, etc.

All the operations of a `pView` must be routed to the underlying collection. To support this, a mapping is needed from elements of the `pView` to elements of the underlying collection. This is done by assigning a unique identifier to each `pView` element (assigned by the `pView` itself); the elements of the collection must also have unique identifiers. Then, the `pView` specifies a *mapping function* from the `pView`'s *domain* (the union of the identifiers of the `pView`'s elements) to the collection's domain (the union of the identifiers of the collection's elements).

More formally, a `pView` $\mathcal{V}$ is a tuple

$$\mathcal{V} \stackrel{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}) \tag{3.1}$$

where $\mathcal{C}$ represents the underlying typed collection, $\mathcal{D}$ defines the domain of $\mathcal{V}$, $\mathcal{F}$ represents the mapping function from $\mathcal{V}$'s domain to $\mathcal{C}$'s domain, and $\mathcal{O}$ is the set of operations provided by $\mathcal{V}$, which must also be supported by $\mathcal{C}$.

Table I.: STAPL `pViews` and corresponding operations.

`transform_view` implements an overridden read operation that returns the value produced by a user specified function, the other operations depends on the `pView` the transform `pView` is applied to. `insert_any` refers to the special operations provided by STAPL `pContainers` that insert elements in unspecified positions.

	read	write	[ ]	begin end	insert erase	insert any
`array_1d_pview`	✔	✔	✔	✔		
`array_1d_ro_pview`	✔		✔	✔		
`static_list_pview`	✔			✔		
`list_view`	✔	✔		✔	✔	✔
`matrix_pview`	✔	✔	✔			
`graph_pview`	✔	✔			✔	✔
`strided_1D_pview`	✔	✔	✔	✔		
`transform_pview`	✔		-	-		
`balanced_pview`	✔		✔	✔		
`overlap_pview`	✔		✔	✔		
`native_pview`	✔		✔	✔		
`repeated_pview`	✔		✔	✔		

Note that we can generate a variety of `pViews` by selecting appropriate components of the tuple. For instance, it becomes straightforward to define a `pView` over a subset of elements of a collection, e.g., a `pView` of a block of a `pMatrix` or a `pView` containing only the even elements of an array. As another example, `pViews` exist that transform one operation into another. This is analogous to backinserter iterators in STL in which a write operation is transformed into a pushback on a container.

Table I shows a list of some `pViews` available in STAPL. These `pViews` are implemented using the schema discussed above, and new `pViews` can be implemented and

created in the same way. The *native* `pView` is a `pView` whose partitioned domain $\mathcal{D}$ matches the data partition of the underlying `pContainer`, allowing data references to it to be local. The *balanced* `pView` partitions the data set into a user specified number of pieces. The sizes of the pieces differ by at most by one. This `pView` can be used to balance the amount of work in a parallel computation. If STAPL algorithms can use balanced or native `pViews`, then performance is greatly enhanced.

III.3. PARAGRAPHs

`PARAGRAPHs` [50] are the building blocks of applications developed using STAPL. They simplify the development of parallel algorithms by allowing a developer to separate the implementation of the individual operations of their algorithm from the specification of the dependences between the operations. They allow developers to compose task graphs that represent their application's required computation.

Definition 1 (Task)**.** *A Task T is a pair (A, D) where A is an algorithm and D is the data that represents the inputs and outputs of A.*

Definition 2 (Task Graph)**.** *A task graph TG is a graph whose vertices are tasks, and whose edges represent data dependences between the tasks.*

In STAPL, the algorithms used to create tasks are referred to as *workfunctions* and the data used by a task is specified by `pViews`, as shown in Figure 3. Workfunctions can be relatively simple, primitive operations such as `plus<T>()`, which has the same behavior as in STL. However, they can also be higher order functions (i.e., receiving other workfunctions as inputs) to implement generic parallel patterns such as `map_reduce`. Tasks with higher order *workfunctions* are referred to as `task factories`, as they define a nested task graph and populate it with tasks and edges.

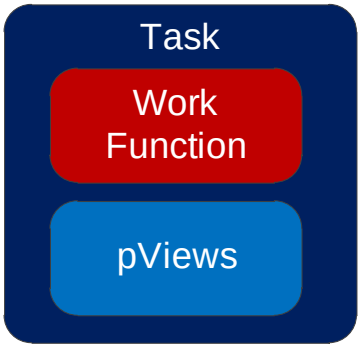


Fig. 3.: A STAPL `PARAGRAPH` task.

Definition 3 (`task factory`)**.** *A* `task factory` *is defined as the tuple* $\{\mathrm{SP}, \mathrm{DP}(O)\}$*, where* SP *is a* structural pattern *and* DP *is a* dependence pattern *that is parameterized by the tuple of operations,* O*, provided to the* `PARAGRAPH`*.*

The *structural pattern*, SP, defines how many tasks are generated for a `PARAGRAPH`, and which of these tasks can be predecessors of tasks in another task graph through task graph composition. For every point in SP the *dependence pattern*, DP, generates a specification of the task that should be created, including the operation from O to be applied, the set of preceding tasks, and the number of successors.

While the `PARAGRAPH Executor` interfaces for implementing workfunctions are described in Section IV.3 (with special attention given to `task factories` in Section IV.3.6), we give two brief examples here of how STAPL users express task graphs in their code. Consider the source code shown in Figure 4 which computes the inner product of two one dimensional views. The workfunction `inner_product_wf` is implemented as a C++ *function object*, and the `map_reduce` `PARAGRAPH` pattern is used. An `inner_product()` freestanding function which calls the workfunction with the view and operation parameters is also provided. It can be used when the `pAlgorithm` is

```
1  template<Op1, Op2>
2  struct inner_product_wf
3  {
4    Op1 op1;
5    Op2 op2;
6    ...
7    template<V1, V2>
8    auto operator()(V1 v1, V2 v2)
9    {
10     return map_reduce(op1, op2, v1, v2);
11   }
12 };
13
14 template<V1, V2, Op1, Op2>
15 auto inner_product(V1 v1, V2 v2, Op1 op1, Op2 op2)
16 {
17   inner_product_wf<Op1, Op2> wf(op1, op2);
18   return wf(v1, v2);
19 }
```

Fig. 4.: The `map_reduce` pattern used to implement `inner_product()`.

not passed as an argument to a higher order function but is instead invoked directly.

The `PARAGRAPH` for the inner product workfunction is shown in Figure 5, with `Op1` set to multiplication and `Op2` set to addition. The return values of the multiplication tasks are passed as input to the first row of addition tasks. The addition tasks form a tree which reduce to an output of a single element, representing the result of the `PARAGRAPH`.

We can now use this inner product implementation as a building block to compose a more complex workfunction, such as the one for matrix-vector multiplication, as shown in Figure 6. Using the `map` pattern (named `map_func` in STAPL to avoid ambiguity with the STL data structure), we employ `inner_product` together with

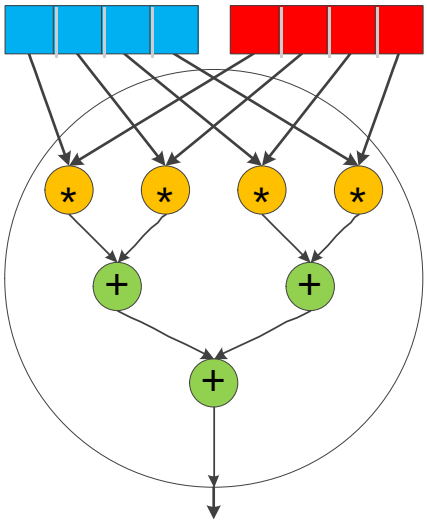


Fig. 5.: The PARAGRAPH generated for `inner_product()`.

The `map_reduce` pattern is employed, and map tasks are denoted with "*" and flow their outputs as input to reduction tasks denoted by "+". The final reduction task produces the result of the PARAGRAPH computation.

both a `row_view` of the 2D matrix and a `repeated_view` of the vector to compute the result 1D vector result.

The writer of `matvec_wf`, perhaps without knowing it, has defined a hierarchical task graph and expressed multiple levels of parallelism. The generated PARAGRAPH shown in Figure 7 demonstrates how a STAPL user can concisely create an exact specification of a relatively complex dependence pattern.

Note that these PARAGRAPH specifications do not address the problem of mapping the task graph onto any real machine for execution. Instead, they express the inherent parallelism of the computation, independent of architectural parameters such as the number of processors. These issues are left to the `PARAGRAPH Executor`, which provides support for PARAGRAPH runtime initialization, task partitioning, inter-task data

```
1  struct matvec_wf
2  {
3    template<2DView, 1DView>
4    auto operator()(2Dview matrix, 1Dview vec)
5    {
6      return map_func(
7          inner_product_wf<multiplies, plus>(),
8          matrix.rows(),
9          repeated_view(vec)
10     );
11   }
12 };
```

Fig. 6.: Composing the matrix-vector multiplication workfunction.

propagation, and locality optimizations. We discuss the interface and implementation of these features in the coming chapters.

III.4. The STAPL Runtime System

The STAPL runtime system (RTS) is the only platform specific component of the library that needs to be ported to each target. It provides a communication and synchronization library (`ARMI`) [57], and a *scheduler* for `PARAGRAPH` tasks. The RTS is not intended to be used directly by the STAPL user.

The RTS provides *locations* as an abstraction of processing elements in a system. A *location* is a component of a parallel machine that has a contiguous address space and has associated execution capabilities (e.g., threads). Different locations can communicate exclusively through `ARMI`, the Adaptive Remote Method Invocation library, which represents the communication layer of the RTS. Special types of objects, named `p_objects`, implement the basic concept of a shared object. The representative of a

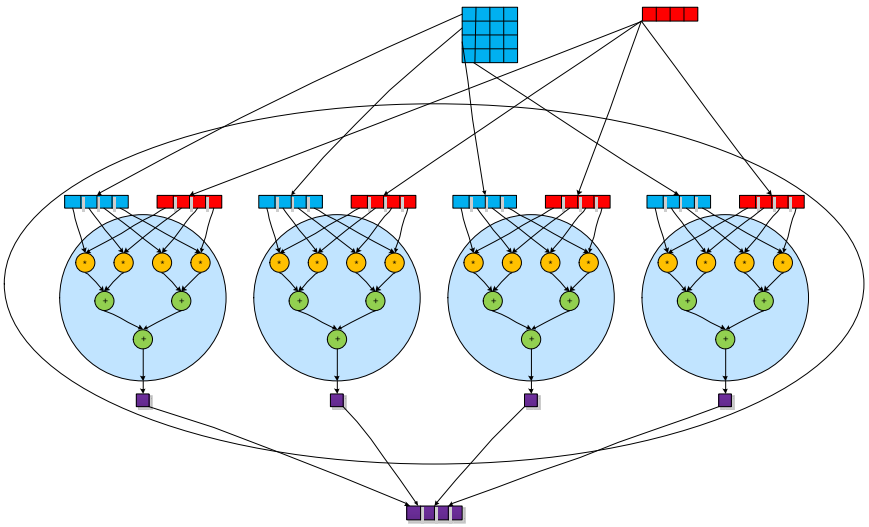


Fig. 7.: The composed `PARAGRAPH` generated for matrix-vector multiplication.
A nested inner product `PARAGRAPH` is created for each row of the matrix and together with a copy of the input vector. The results of these `PARAGRAPHs` are concatenated by `map_func` to form the result vector.

`p_object` in each location has to *register* with the RTS to enable Remote Method Invocations (RMIs) between the representative objects. RMIs enable the exchange of data between locations and the transfer of the computation from one location to another.

RMIs are divided into two classes: *asynchronous* RMIs and *synchronous* RMIs. The former execute a method on a registered object in a remote location without waiting for its termination, while the latter block waiting for the termination of the invoked method. A mechanism is provided to asynchronously execute methods that return values to the caller. As parallel machine sizes reach processor counts into the millions, it becomes essential for algorithms to be implemented using only asynchronous RMIs. In STAPL, these operations implement computation migration, which

allows scalability for very large numbers of processors. We also provide `sync_rmi`s for completeness, but their use is discouraged. The RTS guarantees that requests from a location to another location are executed in order of invocation at the source location.

The RTS provides RMI versions of common aggregate operations. These primitives come in two forms: *one-sided*, in which a single requesting location invokes the execution of a method in all others, eventually receiving a result back, and *collective*, in which all locations participate in the execution of the operation. All the RMI operations, point-to-point, single-sided, and collective, are defined within communication groups, thus enabling nested parallelism. Collective operations have the same semantics as the traditional MPI collective operations. The provided operations include broadcast, reduce, and fence. The fence operation, called `rmi_fence`, when completed, guarantees that no pending RMIs are still executing in the group where it is called.

The RTS provides some optimizations to minimize bandwidth and reduce overhead. The major techniques used are *aggregation*, that packs multiple requests to a given location into a single message, and *combining*, that supports the repetitive execution of the same method in a given location without incurring a large overhead for object construction and function calls. Memory management and the number of messages aggregated are managed by the RTS adaptively according to the application needs.

Another RTS component, the *scheduler*, works with the `PARAGRAPH Executor` to execute tasks created by the `pAlgorithm` specification provided by the `PARAGRAPH`. Task are given to the scheduler once dependences have been satisfied, and it chooses the next task to run by employing a customizable scheduling policy. At this point, tasks can be assigned to execution threads and are considered independent. We will

discuss the scheduler in more detail in Chapter V when we present the implementation of the `PARAGRAPH Executor`.

CHAPTER IV

THE STAPL TASK GRAPH INTERFACE

As stated earlier, a STAPL user is free to employ existing `pAlgorithms` as building blocks when developing a new `pAlgorithm`. This programming model requires that there be support for nested parallelism, as any task that a `PARAGRAPH` creates may itself be a `PARAGRAPH` which describes another parallel task graph. This approach leads to a hierarchical task graph program representation, as shown in Figure 8, that must be efficiently created and executed by STAPL. This runtime administration of `PARAGRAPHs` is the responsibility of the `PARAGRAPH Executor`.

In this chapter, we begin by providing two useful definitions related to values associated with data dependences and then look at the specification of these dependences in a parallel task graph. We formalize the problem, breaking down the creation of a task graph edge into a series of steps. We use this formalization two discuss two previous approaches to the problem, and then define our approach. We then turn our focus to the specification of tasks, describing various workfunctions concepts organized into a taxonomy, giving examples of each. We finally describe annotations that workfunction writers can employ to modify the behavior of input view parameters.

IV.1. Defining Dependence Value Propagation

Tasks receive data associated with a dependence as a parameter, via a `pView`, to their workfunction. In STAPL this information can be propagated between tasks in one of two ways, as illustrated in Figure 9. First, two tasks with a dependence between them can be ecreated with views referring to the same `pContainer`. Changes to elements in the predecessor task's view parameters will be available to successor task via this parameter. In this case, the `PARAGRAPH Executor` is only responsible

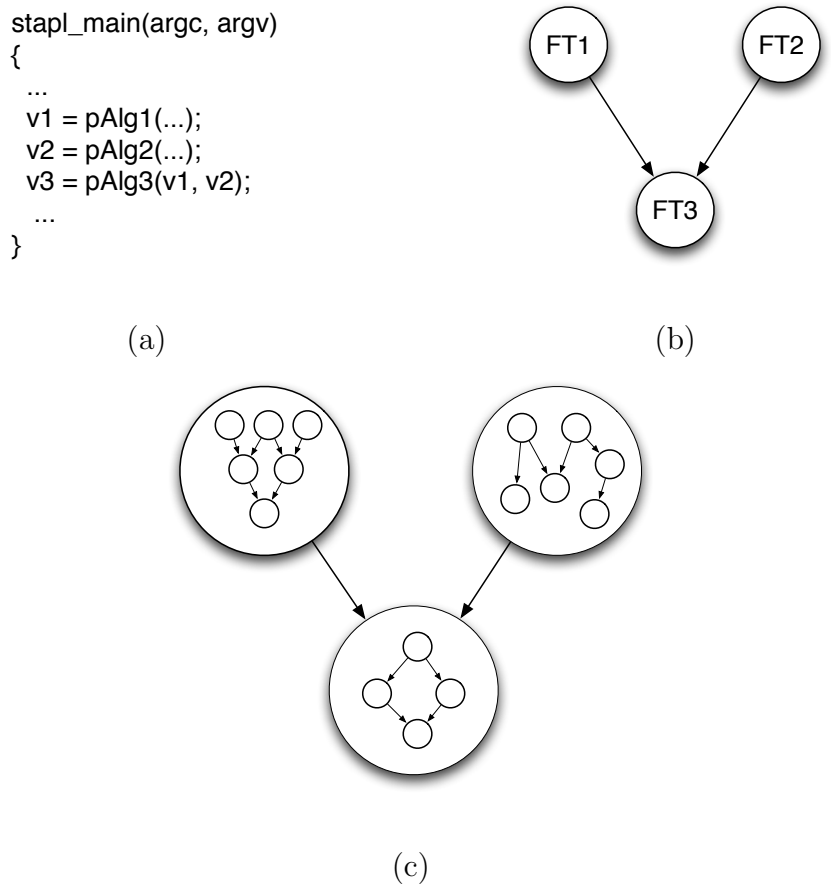
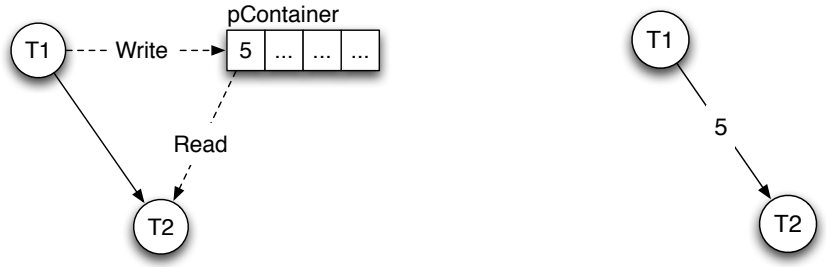


Fig. 8.: `PARAGRAPH` instantiation and task graph execution.

(a) User algorithm invocations instantiate `PARAGRAPHs` which create (b) a graph of factory tasks, each of which corresponds to a nested task graph (c).



(a) `pContainer` value propagation (b) return value propagation

Fig. 9.: Data dependence value propagation.

to *signal* satisfaction of the dependence to the successor and not to *propagate* the intermediate value. Alternatively, intermediate values can be communicated by the *return value* of the workfunction, with propagation occurring in a manner more akin to functional programming [31]. Successors specify their dependence on this value using functions we describe later. In this case, the `PARAGRAPH Executor` must both *signal* the successor task and *propagate* the intermediate value.

Importantly though, regardless of how the value is propagated, the successor's workfunction receives the argument in the same manner (i.e., its formal parameters remain unchanged). In fact, both approaches can be used simultaneously within a task graph. Despite this uniform treatment at the workfunction level, these two approaches require different support from the `PARAGRAPH Executor`. Therefore, in this dissertation when discussing our implementation, we will refer to dependences using the `pContainer` based approach as *signal edges* and the functional approach as *value flowed edges*.

IV.2. Overlapping Graph Creation and Execution

A key contribution of the `PARAGRAPH Executor` is the capability it provides to overlap task graph creation and execution. These two logically separate activities are able to proceed concurrently on each location of the distributed computation. In this section, we present the design decisions of this asynchronous model for the task graph container and discuss its ramifications on the programming interface the `PARAGRAPH Executor` provides to `PARAGRAPH` writers.

IV.2.1. Formalizing the Dependence Specification

Consider the following dependence of task T_2 on task T_1 which we wish to represent in a program:

$$T_1 \rightarrow T_2$$

We can divide each task into the creation activity (T_{C_i}) and the execution activity (T_{Exec_i}). Additionally, we denote the dependence edge creation activity as $T_{Edge_{1,2}}$. Given this notation, we can specify the minimal constraints which define the fundamental partial ordering that must be enforced during the program execution by any implementation as shown in Figure 10.

Clearly, each task's creation must precede it's own execution. Additionally, the tasks' execution activity ordering is simply an enforcement of the original, user specified dependence. Finally, the dependence creation must occur prior to T_{Exec_2}, so that this execution activity can be properly notified of the completion of T_{Exec_1} (and receive any value flowed on the edge). Requiring additional ordering constraints involving $T_{Edge_{1,2}}$ is typically necessary to deal with constraints of real systems. For example, in order to safely retire T_{Exec_1} (destroy values produced and reclaim memory

$$T_{C_1} \rightarrow T_{Exec_1}$$
$$T_{C_2} \rightarrow T_{Exec_2}$$
$$T_{E_1} \rightarrow T_{Eexec_2}$$
$$T_{Edge_{1,2}} \rightarrow T_{Exec_2}$$

Fig. 10.: Minimal ordering constraints in the presence of a dependence.

storing its completion status), the program must know that all successor edges have been created and subsequently traversed (i.e., notified). This implies at least some part of the $T_{Edge_{1,2}}$ activity has completed prior to this retirement, which is typically first considered immediately after T_{Exec_2}.

IV.2.2. Previous Approaches to Dependence Creation

Our design of the task graph is motivated by the limitations of two common approaches to task specification in parallel applications. The first approach is to serialize these activities, usually with a global barrier between creation and execution phases to ensure a globally consistent state. This approach may employ an iterative process (e.g., refine the tasks and dependences between execution timesteps), but these two activities remain serial. Stated more formally, this approach adds the following constraints to those listed in Figure 10:

$$\forall i \, \forall j \;\; T_{C_i} \rightarrow T_{Exec_j}$$
$$\forall i \, \forall j \forall k \;\; T_{Edge_{i,j}} \rightarrow T_{Exec_k}$$

Depending on the implementation, there may be additional restrictions. For example, if one or both vertices must exist in the graph prior to dependence cre-

ation, this must be enforced in some manner. If both must exist, then the following constraint is added, enforcing that all tasks be created prior to the creation of any edges:

$$\forall i \, \forall j \forall k \;\; T_{C_i} \rightarrow T_{Edge_{j,k}}$$

This technique is often employed by large scale, distributed scientific applications. Intel Concurrent Collections [36] [14] and Pegasus [22] also fall in this broad category. While this method is effective for certain classes of relatively static computations and exhibits good scalability within the phases, it suffers several drawbacks:

- **Global synchronizations are an overhead that limit parallelism.** Fundamentally, global synchronizations are used when either insufficient dependence information is available to employ point-to-point synchronization or when enforcing the exact dependences is too taxing on resources (e.g., saturation of the communication subsystem). The former is usually the case, and we will show how the interface to the `PARAGRAPH Executor` addresses this issue, removing the need for this collective synchronization.

- **Dynamic computations are hard to define in this model.** A task graph where sections cannot be created without prior, partial execution require additional effort (i.e., iteration over the two phases).

- **Sufficient memory must be allocated to store the task graph in its entirety prior to the start of execution.** This limits the amount of memory available to the actual computation. As we discuss later, the overlapping of creation and execution enables *incremental generation*, giving STAPL the ability to throttle the memory requirements of the task dependence graph.

A second previous approach, which we refer to as *deferred task creation*, enables new tasks to be created concurrently with task execution. This functionality, however, usually comes at the expense of the direct specification of dependences between tasks. Since no edges are supported in the graph, the $T_{Edgei,j}$ activity is nonexistent in this context. This is not to say that dependences cannot be indirectly specified. Instead the dynamic task creation feature must be employed by the user to defer creating a task until all predecessors have finished executing; typically the last statement in a predecessor is to create the successor task. Stated more formally, for the dependence $T_1 \rightarrow T_2$, the following constraint must be added to those in Figure 10:

$$T_{Exec_1} \rightarrow T_{C_2}$$

This approach has several weaknesses:

- **Level of abstraction of the programming model is lowered.** The specification of the computation a task performs becomes unnecessarily coupled with the expression of how it relates to the overall execution.

- **Critical path is increased.** The critical path of the execution is fundamentally increased by requiring $T_{E_1} \rightarrow T_{C_2}$. Even if we lack the resources to execute these tasks in parallel, this dependence detracts from our ability to better overlap useful work with communication latency. As this latency grows substantially in a large, distributed system, it becomes necessary to allow at least some of T_{C_2} to precede the completion of the predecessor's execution. For this reason, a pure version of this dynamic approach is not typically seen outside shared memory systems with a relatively small degree parallelism.

- **Indirect specification of producer / consumer relationships.** Without explicit dependence edges to flow values between tasks, producer / consumer

relationships are usually handled indirectly via shared storage (i.e., tasks communicate via side effects). In shared memory this can lead to unnecessary locking (due to multiple consumers and memory reuse). In distributed systems, it can lead to memory consistency [3] issues that must be managed, as the value propagation and task creation may occur asynchronously.

Some dynamic task libraries that fall into this broad category, such as current TBB release [32] and Cilk [26], offer some additional, limited support to enforce ordering in the form of of *continuations* or *joins*. These mechanisms allow a parent task to be notified when tasks it created have completed. However, arbitrary dependence graphs still cannot be specified, and the user's programming model becomes even further intertwined with dependence specification.

IV.2.3. Our Approach to Dependence Creation

The key modification made to the previous approaches by the `PARAGRAPH Executor` is to *divide* the edge specification activity ($T_{Edge_{i,j}}$) and *fuse* part of it with with source creation (T_{C_i}) as well as target creation (T_{C_j}). For example, by specifying the out-degree with source creation and the in-degree with target creation, we provide sufficient information to guide the retirement and dependence notification processes, respectively. Using this technique, we allow task creation and execution to proceed concurrently. Below we describe several candidate approaches, based on different divisions of the $T_{Edge_{i,j}}$ activity.

- **Source exact, target count.** In this approach, consumers specify all the tasks on which they depend during task creation, while producers need only specify their out-degree. The task graph implementation must use this information to create the edge, guaranteeing the source task is not retired until the specified

number of targets have created the edge and received notification along with any data flow. Specifically, initialization communication is required from the location where the consumer will execute to the producer's location, requesting it to be notified when this predecessor has completed. Hence, the parameters to task creation take the following form:

$$add\_task(identifier, task, list < source\_id >, out\_degree)$$

- **Source count, target exact.** Here, we reverse the requirements of the previous case, with producers specifying an exact list of consumers, while consumers specify their in-degree. Again, it is left to the implementation to create the edge that enforces dependences and guides task retirement. An additional issue that must be addressed is the order in which input from multiples predecessors should be passed to a task's *workfunction* invocation. While this is explicitly set when the target specifies the exact list (the ordering of the list is the specification of the parameter order), here this ordering must be defined in another manner. For example, source tasks could be required have to specify the pair `<task_id, parameter_index>` for each target during creation. In this case, the parameters to task creation are as follows:

$$add\_task(identifier, task, in\_degree, list < target\_id, paramater\_index >)$$

- **Source exact, target exact.** This approach requires the user to provide complete edge specification twice, when both the source and target of the dependence are created. This precise information enables us to forgo the initialization communication and parameter ordering requirements of the two previous approaches. The yields the following parameter list:

$$add\_task(identifier, task, list < source\_id >, list < target\_id >)$$

- **Source count, target count, separate `add_edge()`.** Alternatively, we can separate exact dependence specification completely from either task's creation. Sources only specify outdegree (to avoid premature retirement) and targets only specify in degree (to disable immediate execution). A third, independent activity, which can be initiated from any location in the distributed computation, specifies the exact dependence using an interface with a form like the following:

$$add\_task(identifier, task, in\_degree, out\_degree)$$

$$add\_edge(source\_id, target\_id, parameter\_index)$$

Of course, this is not an exhaustive list of possible implementations. For example, degree counts could be removed from vertex creation altogether, relying on explicit user action for making tasks runnable and retiring computed values. However, this starts a move towards complete user management of the task graph, something which is neither desirable nor practical in large applications. Such activities are better left to a task dependence system such as the `PARAGRAPH Executor`.

All of the approaches we enumerate above can coexist within STAPL, with users specifying the policy to use for each instantiated task graph. We have chosen to focus our implementation efforts on the *source exact, target count* approach first. We made this decision because it avoids the incoming edge ordering problem, simplifying the external interface. While this approach may require some additional setup communication for the graph, we have found that the overlapped design is effective in hiding the associated latency and is able to maintain good scalability. Later, we describe the implementation of this approach and discuss how the design is affected by the nondeterminism of task placement in a distributed computation.

In the future, we plan to investigate the merits of the other approaches. For example, it would be interesting to see if the *source execute, target execute* specification would further improve scalability. Even if the latency of *source count, target exact* can be completely hidden, removing unnecessary communication will reduce the possibility of network saturation. For heavily used task graph patterns, even marginal performance improvements would warrant the additional, one-time specification effort required of the `PARAGRAPH` writer.

Finally, note that STAPL can also be used to express task graphs in the manner prescribed by the two previous approaches. For example, adding a task with no input edges and an *out_degree* set to 0 causes the `PARAGRAPH Executor` to behave similarly to the *deferred task creation* model. The task can perform computation and then spawn its successor tasks prior to termination, using workfunction interfaces discussed in Section IV.3.4. This flexibility allows seamless interoperability and incremental redesign of existing task graph specifications.

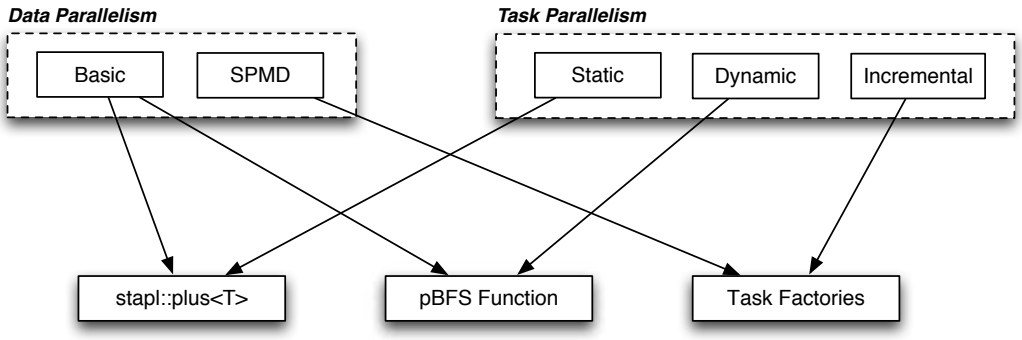


Fig. 11.: Workfunction taxonomy with examples.

IV.3. Task Workfunctions

In the STAPL library, workfunctions define the computation that a task will perform, given the dataset defined by its input views. They also specify how the results of this computation are propagated to the rest of the program, whether it be via side effects (i.e., mutating the input views), the functional approach (i.e., restricting state changes to return values), or a mixture of the two. They are implemented as C++ function objects, with templated function operators. They preferably forgo member data (i.e., all initial state specified via function operator parameters), with incremental workfunctions (Section IV.3.5) being the notable exception.

In this section, we explore the various types of workfunctions available in STAPL, guided by a taxonomy of their different behaviors and providing example usages. We then describe an example *view modifier directive*, which workfunctions can use to define the access requirements of incoming edge data.

The STAPL workfunction taxonomy is shown in Figure 11. Briefly stated, the basic and SPMD concepts vary the initialized degree of parallelism within a task,

with the latter detailing the support for data parallelism. Generally, STAPL users will only create basic workfunctions, with the latter concept mainly existing as an internal building block for `PARAGRAPHs`. Next to be discussed are static and dynamic workfunctions, which define how both *implicit* and *explicit* task parallelism can be created as tasks execute. Incremental workfunctions refine the dynamic workfunction concept and are useful when a workfunction creates a large amount of new, asynchronous work. Finally, task factories bring together several of these concepts and are used in STAPL to define common parallel task graph patterns in `PARAGRAPHs`.

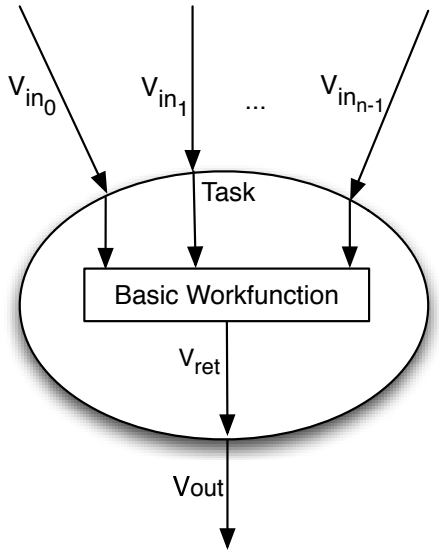


Fig. 12.: Task with a basic workfunction.

```
1  template<T>
2  struct plus
3    // optional, both are default behavior.
4    : public basic_wf, public static_wf
5  {
6    template<typename RefLhs, typename RefRhs>
7    T operator()(RefLhs lhs, RefRhs rhs) const
8    {
9      return lhs + rhs;
10   }
11 };
```

Fig. 13.: Example of a basic workfunction.

IV.3.1. Basic Workfunctions

Figure 12 shows the structure of a task employing a basic workfunction. It has only one explicit thread of computation at the *initiation of execution*. As we will see when we discuss dynamic workfunctions, basic workfunctions are not prohibited from creating additional *explicitly* asynchronous tasks during task execution. They may also *implicitly* specify nested parallelism by invoking a parallel algorithm during execution. An example of simple basic workfunction source code, implementing the STAPL equivalent of `std::plus`, is shown in Figure 13. Note that we use base classes to both tag workfunctions with the taxonomy concepts they adhere to and to provide associated methods. However, in this case it is unnecessary, as *basic* is the default behavior, and it offers no methods to the workfunction writer.

The writers of basic workfunctions are, intrinsically unaware of the locality of their input views. Instead, STAPL chooses a location on which to execute this work, usually in an effort to maximize locality (see Section VI.1).

Table II.: SPMD workfunction types and operations.

Types	**Description**
typedef ... **location_id_t**;	Integral type to enumerate execution logical locations.
Methods	**Description**
location_id_t **get_location_id**();	Location rank identifier of a workfunction instance running inside an SPMD task.
size_t **get_num_locations**();	The number of workfunction instances running concurrently inside an SPMD task.

IV.3.2. SPMD Workfunctions

SPMD workfunctions are the basic building block for data parallelism in STAPL and are primarily used to implement `PARAGRAPH task factories` IV.3.6. Explicit parallelism exists in these tasks from initialization until termination, with multiple *instances* of the workfunction executing concurrently. Both the degree of parallelism and the distribution of these threads on execution locations are not user specified; instead these choices are made by STAPL, the manner in which we discuss below. There are, however, interfaces allowing the user to query information about the logical processors in the task and the data locality of views passed to the workfunction. These interfaces are outlined in Tables II and III.

Table II shows the information provided to the user about the nature of the parallel task, namely the degree of parallelism and the rank of the current workfunction instance. As with all other tasks in STAPL, SPMD tasks represent a separate and distinct *execution context* from other tasks. The logical location running the workfunction instances are placed on physical execution units by the STAPL runtime

system, which also maintains the mapping of this virtual to physical relationship. An important effect of this virtual execution context is that it defines the default locations on which `pContainers` are defined when created in an SPMD workfunction; the data of parallel containers will be distributed among the physical locations on to which logical processors are mapped, in a manner defined by the `pContainer` distribution.

View input arguments to SPMD tasks are passed in full to every concurrent instance of the workfunction, regardless of the data distribution of the underlying container. However, these views are required to provide users some information regarding the locality of elements, as outlined in Table III. This information is key for SPMD workfunctions, as its writer (who has chosen to be directly aware of parallelism) must partition the work explicitly between the concurrent threads, presumably with a goal of minimizing the latency of view element accesses. When invoked within an SPMD workfunction instance, the `local_elements()` method returns a new view over the subset of the elements in the domain of the original input view. This view includes only those elements whose access latency is lower on this instance's physical execution location than any other location with related workfunction instances running. Stated more formally, given the standard view definition:

$$v \equiv (c, d, f_{map}, o) \tag{4.1}$$

with the domain d defined as a finite set of elements,

$$d \equiv \{d_0, d_1, d_2, ..., d_{|d|-1}\} \tag{4.2}$$

and the set of *physical* execution locations Loc_p on to which the SPMD task's *logical* locations Loc_l are mapped (available to the view via the `physical_location()` method of the runtime, which we denote as f_{loc_map}),

$$Loc_l \equiv \{l_{l_0}, l_{l_1}, l_{l_2}, ..., l_{l_{n-1}}\} \xrightarrow{f_{loc\_map}} Loc_p \equiv \{l_{p_0}, l_{p_1}, l_{p_2}, ..., l_{p_{n-1}}\} \tag{4.3}$$

the view, must define a partition of its domain, denoted as D_{part}:

$$D_{part} \equiv \{\{d_a, d_b, d_c\}, \{d_i, d_j, d_k\}, ...\}, \qquad |D_{part}| \equiv Loc_l \tag{4.4}$$

The view accomplishes this by using a locality metric, denoted below as $f_{latency}$. This measure is created in tandem with both the collection and the runtime, and is used to measure the relative cost to access an element from a given location. Using this function, the partition is formed by placing a domain element d_k into the subset D_{part_i}, where i denotes a logical SPMD location and is chosen as follows:

$$\underset{i \in Loc_l}{\text{Min}} f_{latency}(f_{map}(d_k), f_{loc\_map}(i)) \tag{4.5}$$

Hence `view.local_elements()`, when invoked by the SPMD workfunction running on location l_{l_i}, returns a view as defined shown below:

$$v_{local\_elements_i} \equiv (D_{part_i}, d', f'_{map}, o') \tag{4.6}$$

Note that the domain, mapping function, and operations of $v_{local\_elements}$ are left to the view implementor to define as they see fit. Commonly the view has a domain of $[0..|D_{part_i}| - 1]$, an identity mapping function, and is typically limited to read operations.

The locality induced partitioning returned by `local_elements()` is not guaranteed to be exact; a workfunction instance subset may include elements not strictly contained in or nearest to its physical execution location. In fact, in the presence of composed containers, such a strict partitioning of elements may not be possible, as the elements of the view may themselves be views over containers distributed over mul-

Table III.: View types and operations for SPMD workfunctions.

Types	**Description**
typedef ... **locality_aware_view_t**;	Optional type that signals request for view type transformation for locality awareness that will be applied during task creation.
typedef ... **local_domain_view_t**;	View to represent a subset of the domain the this view. Concept of view is implementation defined. Provided by either input view or after locality aware view transformation.
Methods	**Description**
local_domain_view_t **local_elements**(void);	Return view over subset of `domain()`, representing elements that have highest locality with the location running this SPMD workfunction instance. Provided by either input view or after locality aware transformation.

tiple locations. Furthermore, even if possible, maintaining this locality information in the view may itself require computation (i.e., collections with complex, non-closed form data distributions). Hence, when the view deems it infeasible to return precise information, it may return imprecise information, possibly at the expense of reducing the workfunction writer's ability to optimize for latency (task placement mitigates this somewhat, see Section VI.1). In any case, however, the view must provide the SPMD workfunction a proper partition (i.e., each element appears in exactly one instance's `local_elements()` return value).

The view can request aid in computing this locality information, by specifying an optional transformation on itself to be performed as part of task creation process (discussed in more detail in Section V.3.3). If the view defines the type `locality_aware_view_t`, then an instance of it is constructed with the original view

```
1  struct count_five
2    : public spmd_wf
3  {
4    template<typename View>
5    size_t operator()(View view) const
6    {
7      size_t cnt = 0;
8
9      for (size_t &x : view.local_elements())
10       if (view[x] == 5)
11         ++ cnt;
12
13     return cnt;
14   }
15 };
```

Fig. 14.: Example of an SPMD workfunction.

as a parameter. This construction process gathers locality information; and it is this new, transformed view that is passed to the workfunction, with the required `local_elements()` functionality. More formally, given the initial SPMD input view v_i, the function $f_{locality}$ is defined as:

$$f_{locality} \equiv v_i :: locality\_aware\_view\_t \tag{4.7}$$

Applied at task creation, the transformed view passed to the SPMD workfunction, v_{loc_aware}, is defined by the application of $f_{locality}$ that changes the initial set of operations o as shown below:

$$v_i \equiv (c, d, f, o) \xrightarrow{f_{locality}} v_{loc\_aware} \equiv (c, d, f, o \cup local\_elements) \tag{4.8}$$

In Figure 14, we show a simple SPMD workfunction which counts the occurrences

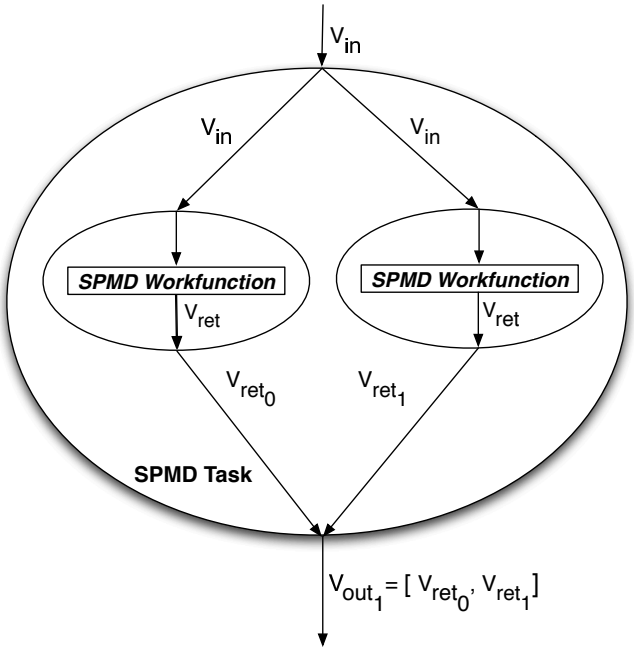


Fig. 15.: Task with an SPMD workfunction.

of a value within the input view. Using `local_elements()`, the work is distributed between the various concurrent instances of the workfunction. The task formed by this specification is shown in Figure 15. Note the handling of the return values from multiple workfunction instances; the output edge of the task is a view over the concatenation of these values.

We have not discussed in detail how the degree of parallelism in an SPMD workfunction is determined. Indeed, this is an area of ongoing work. When an SPMD task is created within a basic task, it employs a single logical thread in the current implementation. This restriction is not an inherent limitation of our design, but rather a simplification employed due to code development time constraints. We discuss the behavior of a nested SPMD invocation in Section IV.3.6. In short, they inherit the

```
1
2  struct substring_count
3    : public basic_wf, public static_wf
4  {
5    template<typename View0, typename View1>
6    auto operator()(View0 str, View1 sub) const
7    {
8      // using stapl::count
9      return count(overlap(str, sub.size()-1), sub);
10   }
11 };
```

Fig. 16.: Example of a static workfunction.

degree of parallelism from the enclosing task. Generally speaking, though, this parameter should be chosen with information from the runtime scheduler (e.g., current processor loads), and information from the view about *current* data locality (data migration may be desired to balance load while minimizing latency). The view, with the help of the underlying container, can provide the set of physical locations on which the elements it references are distributed. Importantly though, due to the abstract interfaces provided to the SPMD workfunction writer, this decision process and the implementation can be refined over time without requiring changes to existing user code.

IV.3.3. Static Workfunctions

Static workfunctions have a logically fixed degree of parallelism from the perspective of their implementors. No interface is provided to this user to explicitly create more concurrent tasks over those initialized when the task was created (e.g., SPMD workfunctions). Similar to basic workfunctions, *static* is the default behavior of STAPL

workfunctions and is assumed so even if no base class is inherited to explicitly select this concept.

Figure 16 shows code for a static workfunction which implements substring occurrence counting. The workfunction writer can remain unaware of the fact that the function `stapl::count()` is actually a parallel algorithm written with the `map_reduce` pattern. As depicted in Figure 17, a nested task graph is transparently spawned (`count()` is expressed as a factory workfunction which we will describe shortly), and its return value is forwarded as the outgoing edge of the static workfunction.

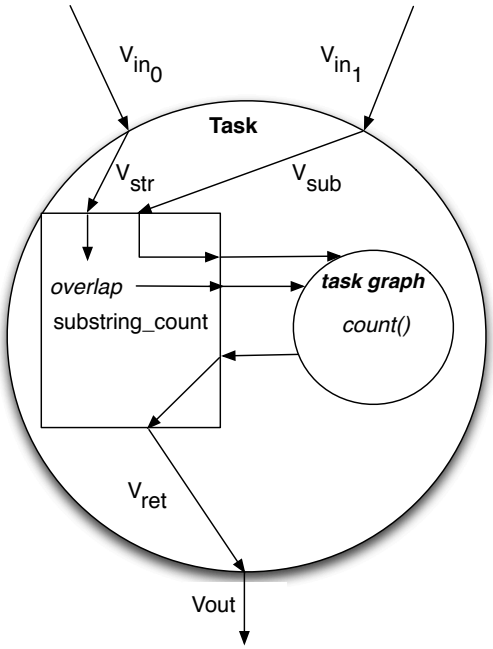


Fig. 17.: Task with a static workfunction.

IV.3.4. Dynamic Workfunctions

Dynamic workfunctions are the fundamental primitive providing task parallelism support in STAPL. They extend the functionality of their static counterparts by giving the workfunction writer the ability to explicitly create new concurrent tasks in the currently running task graph. Both explicit dependences and producer / consumer (i.e., data flow) relationships can be defined for these new tasks. These edges may involve any other task in the task graph, regardless of whether it has been created or executed yet. Furthermore, tasks involved with these edges can be created and executed on different locations.

The public interface of `class dynamic_wf` is summarized in Table IV. Methods are provided to create new tasks as well as provide ingoing and outgoing edge information as described in Section IV.2. Specifically, `add_task` receives the outdegree via the `n_succs` parameter. Furthermore, `consume()` is used to create incoming edges with data flow, while `wait()` specifies dependences with no value to passed to the new task's workfunction.

Figure 18 shows a small example of a dynamic workfunction. Two new tasks are added to the task graph. The first is forwarded the running task's input parameters, while the second receives the values of both the current task and the first spawned task. Figure 19 illustrates the dependence graph created by these operations. Again, note the flexibility in task creation. Namely, the created tasks can depend on any other task in the task graph, regardless of whether or not it has been executed or even created yet. The newly created task can even depend on the task that creates it (as shown in the example in Figure 18). Furthermore, if the user specifies the explicit value of `tid1` (a prototype of `add_task()` exists do so), the order of the two task creation statements can even be switched.

Table IV.: Dynamic workfunction operations.

Types and Metafunctions	**Description**
task_id_t	Indexes the tasks in a task graph.
template ⟨T⟩ edge_reference;	reference type used when specifying consumption of single task (see **consume()**).
template ⟨T⟩ edge_view;	View type for aggregated task consumption.
Methods	**Description**
template ⟨WF, VW0, ...⟩ task_id_t **add_task**(WF, n_succs, VW0, ...);	Create new task in task graph with specified workfunction and views. Set successor count to n_succs. Return task_id of this new task.
task_id_t **task_id**(void)	Return task_id of task executing this workfunction instance.
template⟨T⟩ reference⟨T⟩ **consume**(task_id_t);	Construct reference, representing value by produced by task, suitable as parameter to **add_task()**. Implicitly creates dependence.
template⟨T⟩ array_1D_view⟨T⟩ **consume**(array_1D_view⟨task_id_t⟩);	Similar to above, except specify multiple tasks' values to consume and return a view which aggregates their values together.
implementation defined **wait**(task_id_it);	Passed like view parameter to **add_task()**, explicitly force new task execution to wait until specified task is finished. No value passed to new task's workfunction.
implementation defined **wait**(array_1D_view⟨task_id_t⟩);	Similar to above, instead provide list of tasks to complete prior to executing the new task.

```
1  template<T>
2  struct my_dynamic_wf
3    : public dynamic_wf
4  {
5    template<typename RefLhs, typename RefRhs>
6    T operator()(RefLhs lhs, RefRhs rhs) const
7    {
8      task_id_t tid1 = add_task(plus<T>(), 1, lhs, rhs);
9
10     add_task(
11         multiplies<T>(), 1,
12         consume<T>(task_id()),
13         consume<T>(tid1));
14
15     return minus<T>()(lhs, rhs);
16   }
17 };
```

Fig. 18.: Example of a dynamic workfunction.

IV.3.5. Incremental Workfunctions

Incremental workfunctions refine the dynamic workfunction, allowing a task to divide execution across multiple, successive invocations. The STAPL runtime scheduler will periodically re-invoke an incremental task, querying it after each invocation as to whether or not it has completed all it wishes to do (see the incremental concept in Figure 20). Note that the same effect could be synthesized by using a dynamic workfunction that first did some incremental work and then injected a modified version of itself back into the task graph (via `add_task()`). However, since it is a common pattern in task graph specification, we codify the concept in the taxonomy and allow workfunction writers to explicitly annotate this behavior as it useful information to have when making task scheduling decisions in the runtime system.

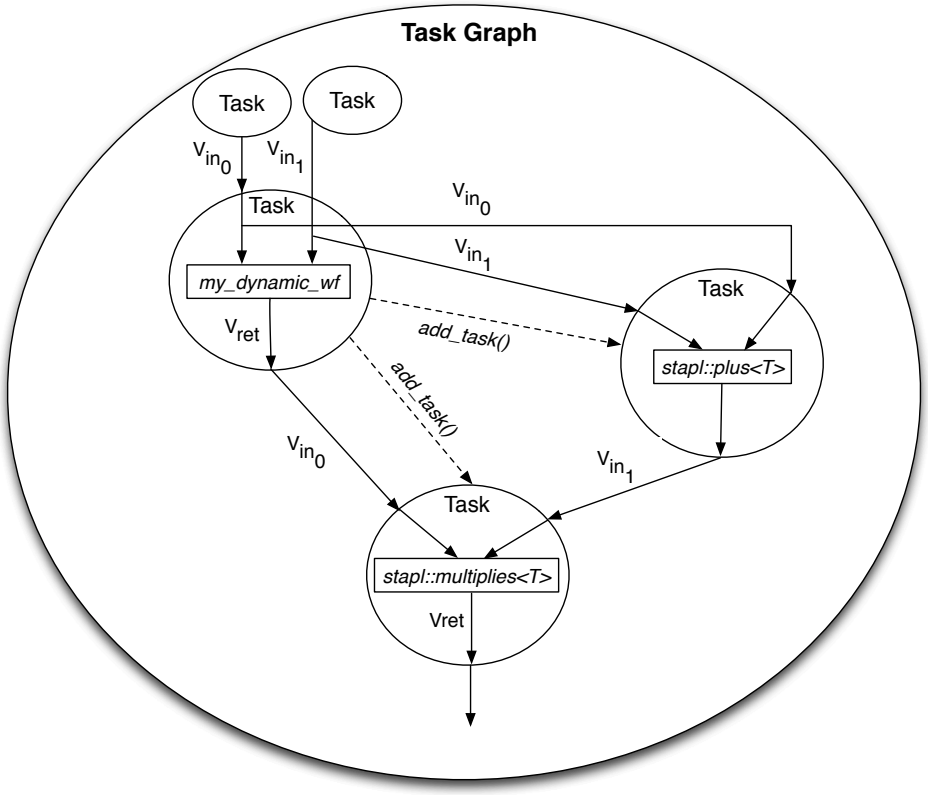


Fig. 19.: A dynamic workfunction creating new tasks.

Incremental workfunctions enable STAPL to manage the amount of a task graph stored in memory. Encapsulated in such a workfunction is the knowledge of the shape of some portion (or all) of the task graph, whose creation can be divided into multiple phases. By design, no guarantee is made as to exactly when each re-invocation happens. Each execution is subject to the scheduler employed by the associated executor. In fact by default, incremental tasks' re-executions are deferred until after all other tasks have been finished, in an effort to minimize the memory overhead of the task graph. However, some task schedulers may choose to re-invoke more often, to increase the window of the task graph it has available for better optimization and

```
1  struct incremental_wf
2    : public dynamic_wf
3  {
4    bool finished(void) const;
5  };
```

Fig. 20.: The incremental workfunction concept.

ordering decisions.

While incremental workfunctions typically exist primarily to spawn additional tasks (that will produce data for consumers further down in the task graph), it is valid for incremental tasks to have outgoing edges, with other tasks in the task graph depending directly on them. When this occurs, these successors will not executed until all re-invocations of the incremental task have completed. If produced data is flowed to consumers, the return value of the last invocation of the workfunction is the instance of this data which will be propagated to consumers.

In Figure 21, we show the implementation of a simple incremental workfunction. The task will be invoked twice, creating a total of three new tasks forming a small section of a task graph as illustrated in Figure 22. In addition to the `finished()` member function, there is another key difference with the workfunctions we have previously discussed, namely the presence of data members at the mutable property of the function operator (e.g., it is not const-qualified). This allows successive invocation of the workfunction to store some state about what it has done so far in previous executions.

Even in this trivial example, the utility of incremental workfunctions is clear. This one task encodes the work and dependence information of four tasks (i.e., itself

```
1  struct my_inc_wf
2    : public incremental_wf
3  {
4    int           m_x;
5    task_id_t     m_last_tid;
6
7    my_inc_wf(task_id_t t0)
8      : m_x(0), m_last_tid(t0)
9    { }
10
11   bool finished(void) const
12   { return m_x == 2; }
13
14   template<typename View>
15   typename View::value_type
16   operator()(View vw)
17   {
18     typedef typename View::value_type value_t;
19
20     m_last_tid = add_task(plus<value_t>(), 1,
21                           vw[m_x],
22                           consume<value_t>(m_last_tid));
23
24     if (++m_x == 2)
25       add_task(multiples<value_t>(), 1,
26                consume<value_t>(this->task_id()),
27                consume<value_t>(m_last_tid));
28
29     return vw[m_x];
30   }
31 };
```

Fig. 21.: Example of an incremental workfunction.

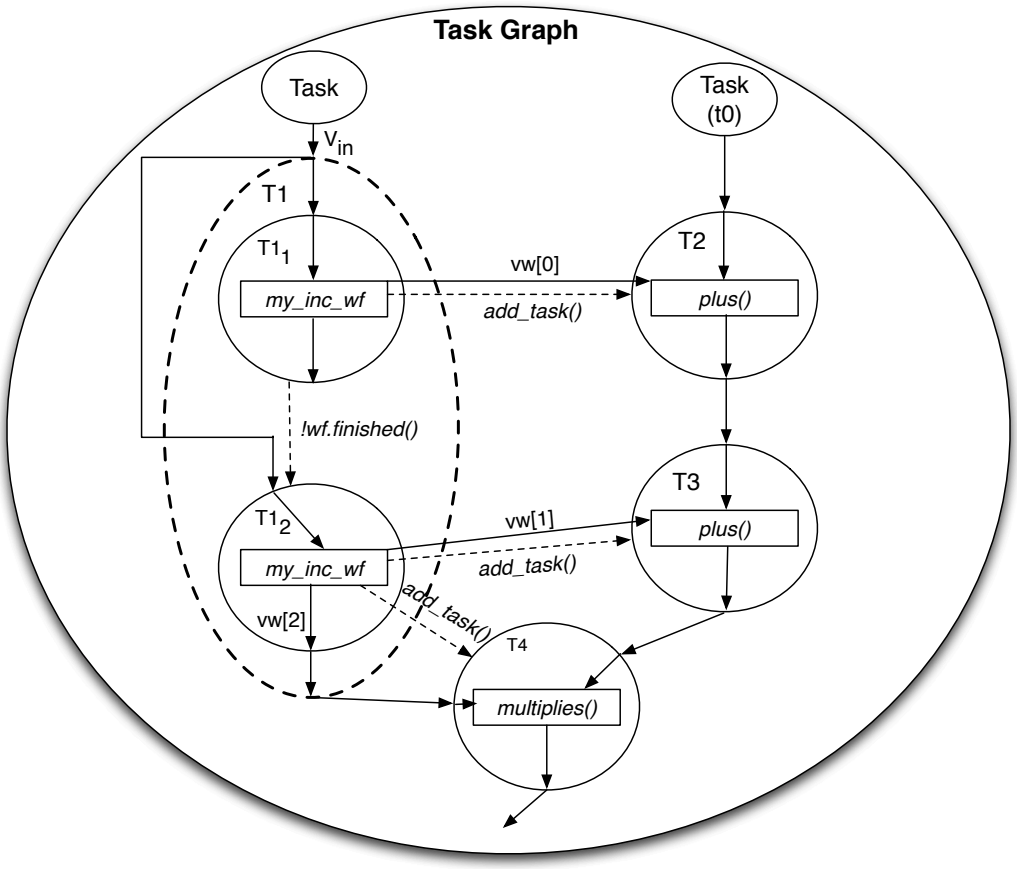


Fig. 22.: Two successive invocations of an incremental workfunction.

and the three spawned tasks), and reduces the memory overhead of the task graph by deferring their creation until this section of the graph is reached. Despite this benefit, no restrictions are placed on the dependence edges for these newly formed tasks; they may be both producers and consumers for other tasks anywhere in the task graph, both from within and outside the newly created subgraph. In the next section, we show how task factories use this concept to incrementally create and execute entire task graphs distributed across multiple execution locations.

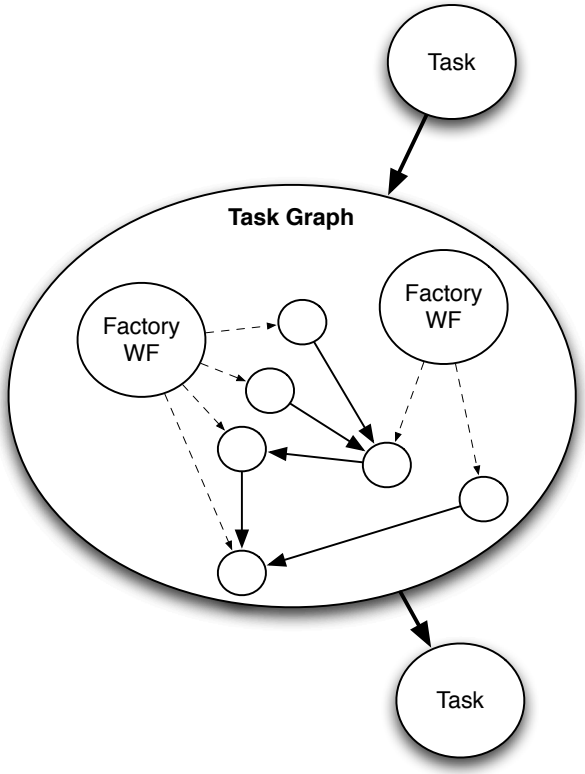


Fig. 23.: The task factory creating a nested task graph.

IV.3.6. Task Factories

Task factory workfunctions combine the functionality of SPMD and dynamic workfunction concepts, yielding a powerful tool for distributing the initialization and execution of task graphs in STAPL programs. Factories encapsulate common parallel programming patterns (e.g., reduce, prefix scan, etc.), and are responsible for creating the runtime tasks that implement their behavior. Furthermore, factories can implement the incremental workfunction concept, enabling the distributed computation to begin execution without requiring the entire task graph to be initialized.

```
1 struct task_factory_wf
2   : public spmd_wf,
3     public incremental_wf
4 {
5   void set_result_task(task_id_t) const;
6 };
```

Fig. 24.: The task factory concept.

Since factories typically spawn a large number of tasks that are logically related, they serve as a natural place to transition from one level in a STAPL program's hierarchical task graph representation to another. For the `PARAGRAPH Executor` user interface, the primary effect is task *identifier scoping* semantics. Usually, newly spawned tasks are assigned task identifiers in the scope of its parent. Similarly, if this new spawned task is itself dynamic, then its children will exist in this same identifier scope. In contrast, factories logically exist in two identifier scopes. At the level at which they are spawned, they appear as any other task, consuming data via input edges and producing values on an output edge. However, the factory is also the first task in a new identifier scope in which all of its descendants (with regard to initialization, not program dependence) are encompassed, as shown in Figure 23.

One feature unique to factories is *return value delegation*. As the progenitor task of all tasks in a STAPL task graph, the factory defines the type of the outgoing edge of the graph (i.e., the edge visible to the level above). However, factories designate another task as producer for the value of this edge, using the interface shown in Figure 24. Again, the primary aim of the factory is to succinctly encode the structure and incrementally generate the pattern of tasks representing the computation. Therefore, it is natural to give it the ability to delegate the production of the graph's outgoing

edge to a descendant. Otherwise, semantics to join one or more spawned tasks back to the factory task prior to its termination would need to exist, adding undesirable complexity to the task graph interface.

A factory specifies this delegation by invoking `set_result_task()` member function. Invoked once per per location where a factory workfunction executes, the method receives a task identifier denoting which task's outgoing edge will represent that instance's return value (per the SPMD concept definition, the factory's outgoing edge is a view over the return value of the concurrent workfunction instances). As each instance may be invoked incrementally (i.e., multiple times), the call to `set_result_task()` can take place in any iteration and different iterations for each workfunction instance. Furthermore, no restrictions are placed on the task chosen by each instance. An instance may specify any spawned task in the graph (even if spawned by another factory workfunction instance). Multiple workfunction instances may also delegate to the same task. This delegation process is illustrated in the example factory listed in Figure 25.

IV.3.7. View Access Specifiers

One of the design goals of STAPL is to facilitate the flow of contextual information from the user, in this case the workfunction developer, to the components it employs in library. *View access specifiers* are one example, allowing writers to annotate their workfunctions with how the input view parameters will be accessed (e.g., read-only, write-only, etc). During the initialization of tasks (see Section V.3.3), we notify the view of this restriction, allowing it to optimize itself for the given pattern. Furthermore, we use the directive to refine our placement of task in the system for execution, as described in Section VI.1.

In Figure 26, we shown example workfunctions employing view access specifiers.

```
1  struct my_factory_wf
2    : public task_factory_wf
3  {
4    int           m_x;
5    task_id_t     m_last_tid;
6
7    my_factory_wf() : m_x(0) { }
8
9    bool finished(void) const
10   { return m_x == 2; }
11
12   template<typename View>
13   void
14   operator()(View vw)
15   {
16     typedef typename View::value_type value_t;
17
18     size_t index = vw.local_elements[m_x];
19
20     // first incremental invocation
21     if (++m_x == 1)
22     {
23       m_last_tid = add_task(foo<value_t>(), 1, vw[index]);
24       return;
25    }
26
27     // second incremental invocation
28     m_last _tid =
29        add_task(multiples<value_t>(), 2,
30                 consume<value_t>(vw[index]),
31                 consume<value_t>(m_last_tid));
32
33     set_result_task(m_last_tid);
34   }
35 };
```

Fig. 25.: Example of a factory workfunction.

```
 1 struct Read {};
 2 struct Write {};
 3 struct ReadWrite {};
 4
 5 struct wf1
 6 {
 7   typedef access_list<Read,Write> view_access_types;
 8
 9   template<typename ReadView, typename WriteView)
10   int operator()(ReadView rview, WriteView wview)
11   {...}
12 };
13
14 struct wf2 : public ro_binary
15 {...};
```

Fig. 26.: Workfunction with view access specifiers.

`wf1` takes the more explicit form of the typedef, while `wf2` employs a class inheritance which can be used as a shorthand for common cases. In addition to any internal optimizations, views also restrict their public interface, disabling any methods that do not adhere to the requested access type.

CHAPTER V

THE STAPL TASK GRAPH IMPLEMENTATION

Now that we have described both the design motivations and the interface of the `PARAGRAPH Executor`, we present in detail how this framework is constructed. We begin by considering several observations that influence the implementation approach we have chosen.

`PARAGRAPH` execution is data driven. Tasks are primarily accessed as part the execution graph traversal. Once a task is inserted into the graph, it need not be accessed again until it can be scheduled for execution (i.e., all incoming edges are available). Therefore, it is not individual tasks that drive this traversal, but rather the *flow of values* along dependence edges. This insight leads us to the fact that our primary data structure, the *edge container*, provides random access to the intermediate values the task graph has produced (or will produce) and the associated consumer metadata. Vertices (i.e., tasks) are referred to in edge container entries, but need no explicit container themselves to provide independent access.

Forward progress in the execution traversal is enabled by a small number of asynchronous events. Executing a task makes outgoing values available, which enable mores tasks to execute, and may also spawn additional tasks. Inserting tasks enable notification from predecessors to complete, allowing recycling of resources from previous computations. Events such as these occur asynchronously in the local *initialization* and *execution* and may also be initiated from incoming *remote method invocations* from other location participating in the computation. With this behavior in mind, our implementation is driven by such events that serve to transition tasks in the system through a series of *lifetime states*, described in Section V.3.1, that begin with *uninitialized* tasks and end with task *retirement*.

The lifetime of values produced by a task graph exceeds that of the computation that produced it. After a task completes execution, most of the resources it used can be immediately reclaimed by the system. Only the value it produces and some metadata about its consumers need be maintained. This remaining state can be reclaimed once all consumers have completed execution. This observation suggests that the various components within the task graph should have lifetimes that are determined independently.

The creation and execution of a task occur on locations known only at runtime. The location partitioning of both *task creation* (i.e., `add_task()` invocations) and *task execution* is influenced by user level algorithms and data distributions, of which the `PARAGRAPH Executor` may have no knowledge of prior to program execution. Furthermore, no ordering with another task's activities can be assumed, other than the fundamental constraints outlined in Section IV.2. Therefore, our implementation provides a mechanism for producers and consumers on different, unknown locations to *asynchronously collaborate* in the dependence edge creation process and any related data flow operations (discussed in Section V.4).

With these design principles in mind, we begin our discussion of the `PARAGRAPH Executor` implementation in the next section by summarizing the major components of the infrastructure. We follow this overview with a description of each component in more detail and illustrate how they interact to efficiently instantiate and execute STAPL `PARAGRAPHs` defined by the application developer.

V.1. PARAGRAPH Executor Infrastructure Overview

The major components of the `PARAGRAPH Executor` infrastructure and their interactions are depicted in Figure 27. The *graph manager* is initialized with the factory

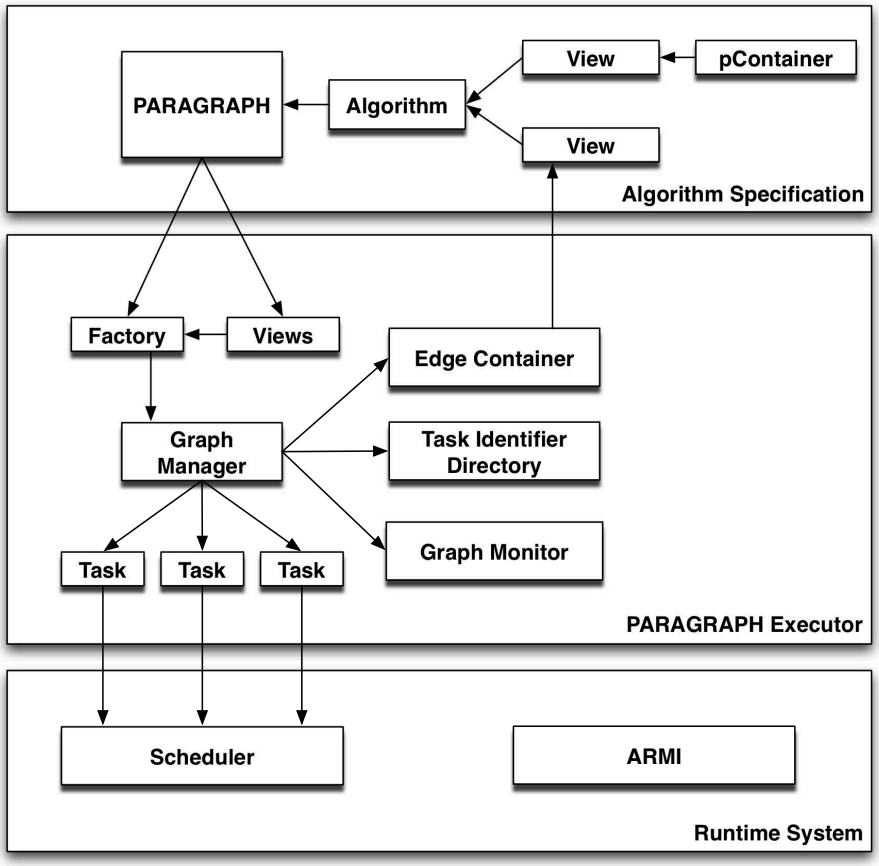


Fig. 27.: Task graph component architecture.

task that the PARAGRAPH has distilled from the user's algorithm and view inputs. These views are either defined over explicitly created pContainers or are backed by values propagated on a previous PARAGRAPH's outgoing edge. The manager creates instances of a *graph monitor* to collect statistics about execution, a *task identifier directory* to aid in the discovery of the task distribution among locations, and an *edge container*. The edge container maintains dependence relationships between tasks, stores edge flowed values, and provides the mechanisms to convey these values to consumer tasks. Finally, the graph manager signals the runtime system to create a new scheduler for this graph. The scheduler is responsible for managing the execution of runnable tasks and subsequently determining when all created tasks have been serviced, signaling global termination of the task graph's computation.

V.2. Graph Manager

In Figure 28, we summarize the graph manager class' major interactions with other components in the PARAGRAPH Executor. There are two interfaces to higher levels of STAPL. First, the PARAGRAPH invokes the graph manger's function operator to initialize a factory task for execution, with an optional customized task scheduling policy. This task uses the factory workfunction created by the PARAGRAPH and the input views associated with this computation. The PARAGRAPH Executor returns a view, as described in Section IV.3.2, which refers to the result values (i.e., outgoing edge) that will be computed by the execution of the task graph.

The second external interface to the PARAGRAPH Executor is the task creation facilities provided to dynamic workfunctions (see Section IV.3) such as factories. This is accomplished by the task_graph_access class, which serves as a base class for all dynamic workfunctions and provides the interface listed in Table IV. Prior to

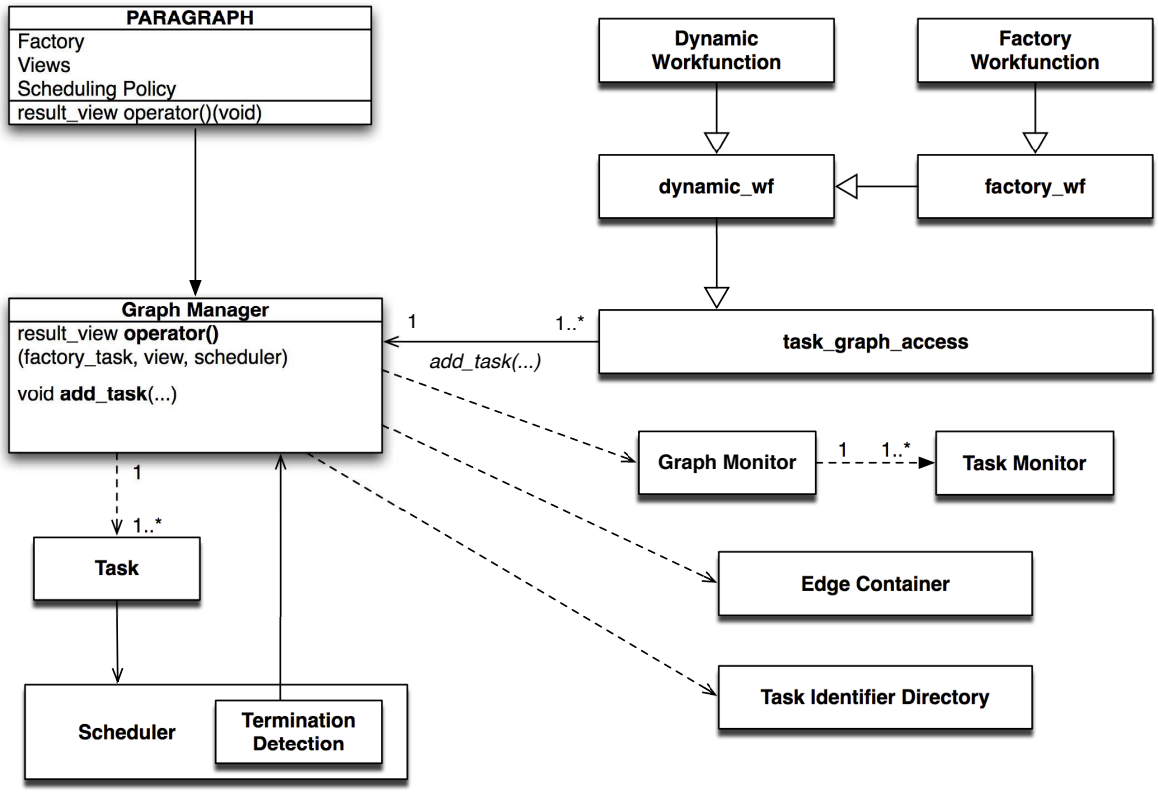


Fig. 28.: The `graph_manager` class and related classes. Lines with clear triangles represent class inheritance, while those with filled triangles represent class interactions. Dashed lines denote data members of a class with labels marking any possible multiplicity in the relationship.

relaying `add_task()` requests to the graph manager, `task_graph_access` normalizes the parameter list to the following form:

- `TaskID` - the task identifier assigned to this task either directly by the dynamic workfunction or internally by the `PARAGRAPH Executor`.
- `WF` - the workfunction specified by the user.
- `VS` - a *ViewSet* collects all input parameters to the workfunction in a heterogeneous container (i.e., tuple [6]). This arrangement simplifies calculations and transformations we will apply on the input during task creation.
- `ExplEdge` - a list of task identifiers enumerating tasks that this task explicitly depends on via `wait()` statements.
- `n_succs` - the number of successor (explicit and data consumers) of the task.

When invoked by the `PARAGRAPH`, the graph manager initializes several structures (denoted by dashed lines) including the *edge container* and the *task identifier directory*, which are discussed in more detail in Sections V.5 and V.4, respectively. Another of these structures is the *graph monitor*. It collects statistics about the graph's execution such as task creation counts, execution times, and `ARMI` usage. The lifetime of these three components are tied to the graph manager; it destroys them following task graph completion.

In addition to providing interfaces to `PARAGRAPHs` and user workfunctions, as well as managing the other components of the `PARAGRAPH Executor`, the graph manager also collaborates with the runtime system's *scheduler*. The scheduler determines the execution ordering of runnable tasks (i.e., tasks with all predecessor values available) as we will discuss in Section V.6. The `PARAGRAPH Executor` also works with the scheduler to determine when all tasks in the distributed execution of the `PARAGRAPH` have

completed, allowing successors `PARAGRAPHs` to be initialized and current `PARAGRAPH Executor` component resources to be released. We summarize this approach to termination detection in Section V.6.1.

V.3. Tasks

There are three primary user interactions with tasks currently in STAPL. First, users create them via `add_task()` calls. Next, when a task is executed, the workfunction specified by the user is invoked. Finally, a task can be referenced via its *task identifier*, when another task is created, to specify an edge in the task graph between them. In this section, we outline what happens internally in the `PARAGRAPH Executor` when the first two interactions occur, deferring discussion of the third until we discuss the edge container in Section V.5. We also describe the primary classes and functions used in our implementation.

V.3.1. Task States

We define five task states that are used internally in the the `PARAGRAPH Executor` to track the lifetime of a task. As previously stated, the design of the task graph is event driven; and events, such as the user interactions mentioned above, enable a task to transition from one state to another. The states a task can be in during its lifetime are depicted in Figure 29, and the individual states are described below. In the sections that follow, we will outline the various events and the subsequent state transitions they cause.

- **Not Initialized (NOT_INIT)** - A task in this state will be created via `add_task()` during a task graph's execution but has not been yet. Both value propagating and simple signal (Chapter IV) successor tasks may refer to it during their cre-

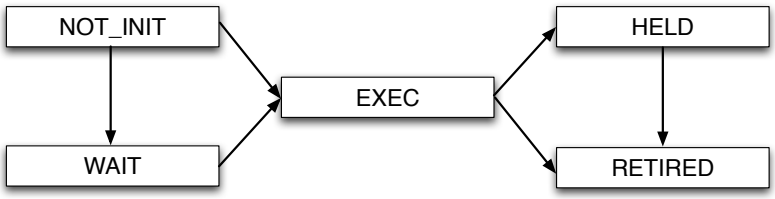


Fig. 29.: Task states.

ation, which due to our relaxed ordering constraints of task specification, may validly occur prior to this predecessor's initialization.

- **Waiting** (WAIT) - Waiting tasks have been created but are not runnable and have not been released to the scheduler. These are tasks having predecessor tasks who have not notified them of completion.

- **Executing** (EXEC) - Executing tasks have had all predecessor tasks complete and have been subsequently notified by them. These tasks are not necessarily currently running, but have been released to the scheduler for execution.

- **Held** (HELD) - Held tasks have finished execution and notified their associative task graph of this completion but have not been completely removed from the graph because they have successors in the NOT_INIT phase that still must be serviced. At a minimum, information signifying they have completed execution together with a copy of their produced values are maintained for consumers that have not been created yet.

- **Retired** (RETIRED) - Retired tasks have finished execution and notified all successors. All related structures can be deleted, and the task is completely removed from the task graph.

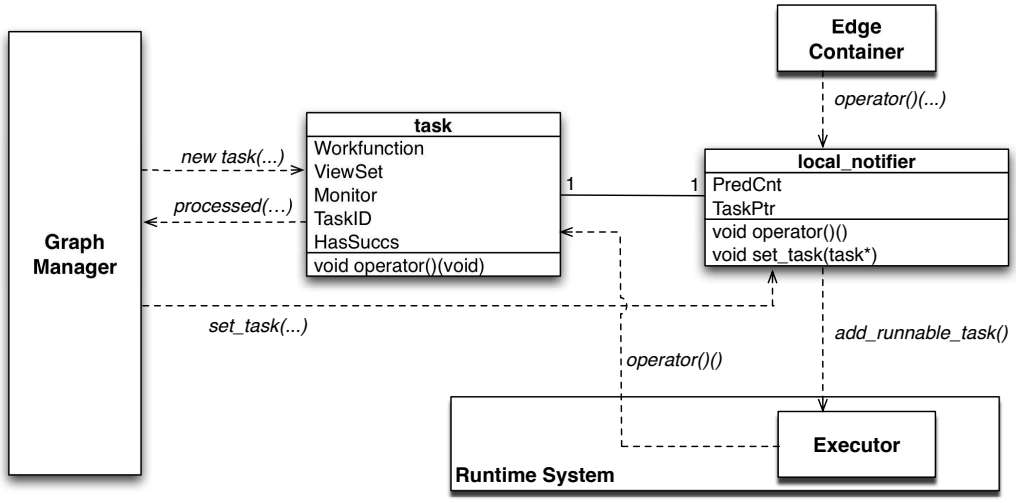


Fig. 30.: The `task` class and related classes.

V.3.2. Task Classes

There are three classes instantiated when a task is created. These classes and their interactions with other components are shown in Figure 30. The class template `task` is parameterized with the workfunction, viewset, and task monitor types. These define data members storing these primary pieces of the task. Additional data members include a numeric task identifier and a boolean stating whether or not it has successor edges in the task graph (i.e., $n\_succs > 0$). The one public method of `task` is the function operator, which is invoked by the scheduler when the task is ready for execution.

The `local_notifier` is a callback function object registered with edge entries in the *edge container* so that it can notify a task (through successive function operator invocations) as edges become available. As discussed in the next section, when predecessors exist, the notifier is created early in task creation, initialized with the in-degree

of the task. Once the `task` object is fully constructed, the notifier is given a pointer to it, and takes responsibility for its eventual release to the scheduler. The object is destroyed when a task's edges are all satisfied and it is passed to the scheduler.

V.3.3. Task Creation

The sequence of actions that occur when `add_task()` is called is illustrated in Figure 31. We begin by incrementing a counter which tracks the number of tasks added to the graph at this location. This aids in the general termination detection approached discussed in Section V.6.1. Next, we determine whether this task should be executed here or migrated to another location by applying the task placement policy which we will describe later in Section VI.1. If the task must be migrated, we use `ARMI` to invoke `add_task()` on that location and return immediately; all responsibility to create, execute, and track the task is transferred to the new location. Note that the new location will reapply the placement policy; a task may be migrated multiple times, gathering more view locality information at intermediate locations, prior to arrival at its execution location where it will be initialized by the `PARAGRAPH Executor`.

Once the task has arrived at its execution location, we call `add_producer()` on the graph's edge container if it has any successors (i.e., `n_succcs` > 0). While we discuss this function in more detail in Section V.5, for now it is sufficient to know the two major actions it performs. First, it allocates local storage both the tasks' return value (i.e., outgoing edge) and its metadata (successor locations, number of uncreated successors, etc). In addition, it registers this task identifier with the identifier directory (see Section V.4), so that its successors can send requests to it.

The next step in task creation is to detect all predecessors of the task. While the explicit dependence edges are available directly in `ExplEdge`, those specified via `consume()` appear as views in the *ViewSet*. To detect these, we traverse the *ViewSet*,

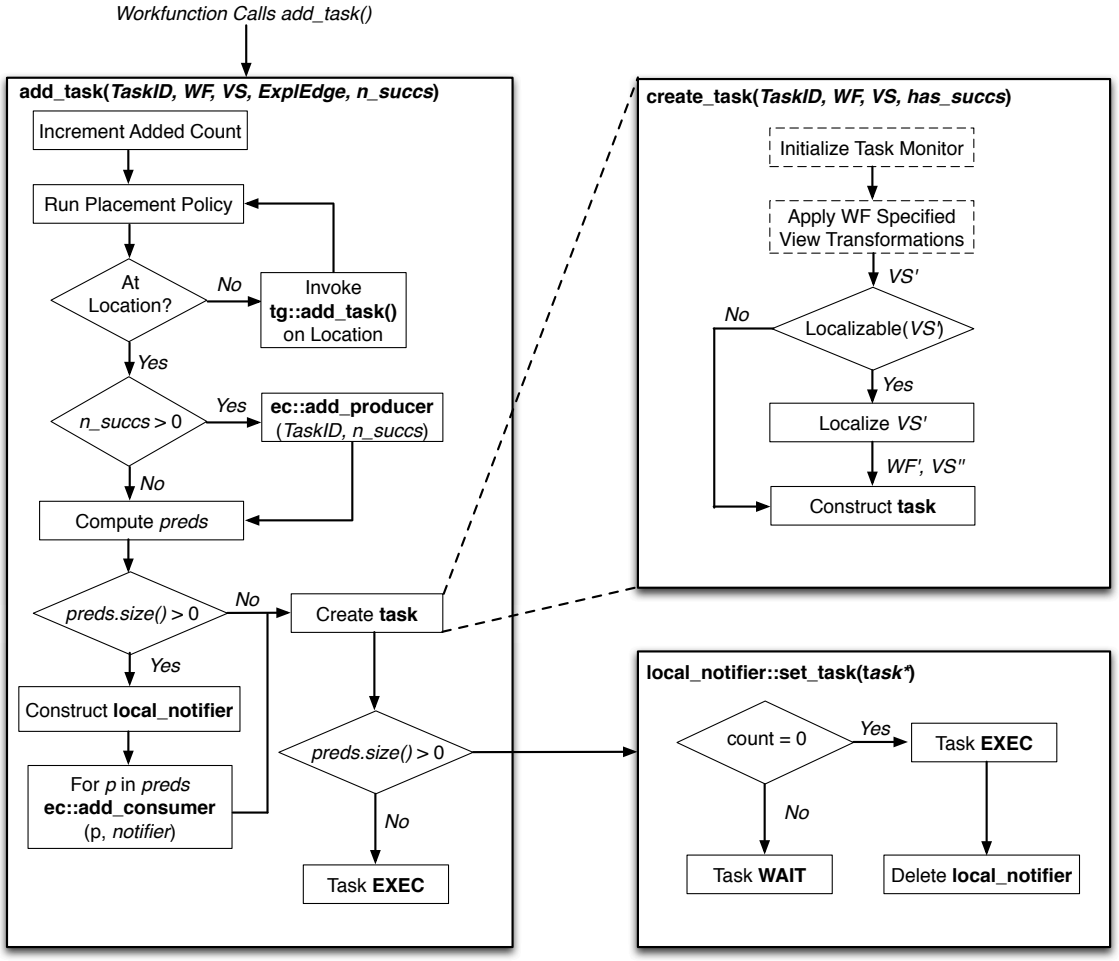


Fig. 31.: The task creation process. Diagrams for the `add_task()`, `create_task()`, and `set_task()` functions.

scanning embedded type information which denotes them as representing values produced by other tasks' return values. We then extract the associated task identifiers, adding them to our predecessor list. If no predecessors exist, we proceed to construct a `task` object, as described next, and then immediately pass it to the scheduler. When this event occurs, the task is transitioned directly from the NOT_INIT state directly to the EXEC state.

V.3.3.1. Creating the `task` Object

The creation of the `task` object is summarized in Figure 31 in the `create_task()` flowchart. First, we initialize an instance of the `task_monitor` associated with this task graph. We next apply any view transformations that have been requested by the workfunction. For example, these transformations may include access modification as described in Section IV.3.7) or the locality awareness transformation discussed in Section IV.3.2.

The final step before actually constructing `task` is *view localization*. View localization attempts to lower the average time to access elements in a view by detecting at runtime when all elements in its collection are contained within the task's physical execution location. If this condition holds, a transformation is applied, which allows the view more efficient, direct access to the collection's elements. View localization is described in detail in Section VI.2; for now it is sufficient to note it can induce both a transformation of view type and workfunction type.

Following this preliminary initialization and input transformation, we are now ready to allocate the `task` object. We construct it with the monitor as well as the post transformation workfunction and viewset. We also initialize a boolean, `HasSuccs`, stating whether its out degree is nonzero. We later use this bit to guard unnecessary calls to the edge container when the task executes. This simple optimization has

dramatic performance benefits in task graph usage patterns where task dependences are enforced by requiring the ordering constraint $T_{E_1} \rightarrow T_{C_2}$, as we discussed in Section IV.2.2.

V.3.3.2. The `local_notifier` Object

If any predecessors exist for a task, we construct a `local_notifier` object initializing a data member `PredCnt` with the task's edge in-degree. For each predecessor, we invoke `add_consumer()` on the edge container, passing the associated task identifier and this newly created notifier. Internally, the edge container is responsible for facilitating value flow and will repeatedly invoke this notifier's `operator()`, once after each incoming edge is available (see Section V.5). Each time, `PredCnt` is decremented.

After creating the notifier, `add_task()` continues by creating the `task` object as previously described. After this process concludes, `add_task()` informs the notifier of this event via the `set_task()` method, passing it a pointer to the `task` object. As both local task execution and `ARMI` message handling are assumed asynchronous from this task creation activity, it is possible that the notifier has been invoked for all edge events (i.e., `PredCnt` = 0). If this is the case, the task is immediately forwarded to the scheduler (once again moving directly from NOT_INIT to EXEC), and this notifier object is destroyed. Alternatively, if `PredCnt` is still nonzero, the notifier stores this `task` pointer and assumes responsibility for the task, which enters the WAIT state. The task will then transition to EXEC when the `local_notifier` receives all pending notifications and releases it to the scheduler.

V.3.4. Task Execution

At some point during the program run, each task in the EXEC state is scheduled by the executor and `task::operator()` is invoked, initiating a sequence of operations

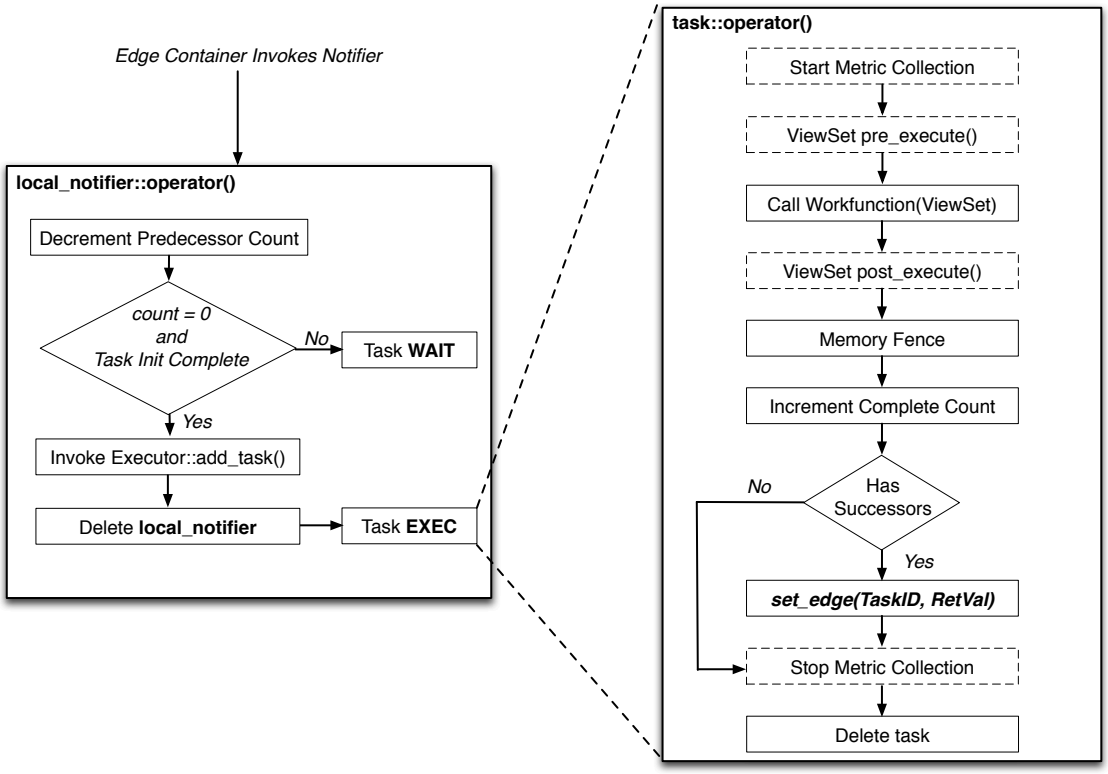


Fig. 32.: The task execution process.

shown in Figure 32. First the task monitor is notified that execution of this task has begun. Next, the tasks' views are notified that execution is about to begin via the optional view method `pre_execute()`. Next, the user's workfunction function operator is invoked, passing the views as arguments to this call. After the workfunction completes, views are once again notified similarly via another optional callback to `post_execute()`.

The next step in the execution sequence, is a memory fence ensuring that any side-effects on parameters (i.e., view writes) have been properly propagated through the system. This operation, together with the two callbacks to the view described in

the previous paragraph, are used to enforce the memory consistency in the underlying `pContainer` between signal dependent tasks (described in Chapter IV introduction) in the graph. The use of these operations and a detailed discussion of `pContainer` memory consistency models are presented in [55].

The `task` invocation next notifies the `PARAGRAPH Executor` that the task has completed. For termination detection, an internal count of the number of tasks that have been executed on this location is incremented. If the task has successors (known by the `HasSuccs` variable), the next step is to pass the value of the task's outgoing edge to the edge container via `set_edge()`, so that this information can be disseminated to them. The task monitor is then notified the task has completed. Finally, the `task` object destroys itself, freeing any memory allocated to it.

After finishing execution, the task transitions from the EXEC state to either the HELD or RETIRED state. The task is removed from the scheduler, its associated `task` object having been deallocated. If the task had no successors, then all operations necessary have been done, and it is RETIRED. Otherwise, the decision of what state to move to is delegated to the edge container. How this process occurs is described in Section V.5.

V.4. The Task Identifier Directory

The task identifier directory helps the `PARAGRAPH Executor` manage the difficulties presented by an unknown distribution of tasks across execution locations, inherent in dynamic applications where such decisions are input dependent (see Section VI.1). Specifically, the *edge container* requires communication between target and source task locations to create edges in the task graph, and their exact locations are not readily known to each other. The `PARAGRAPH Executor` delegates to the directory the

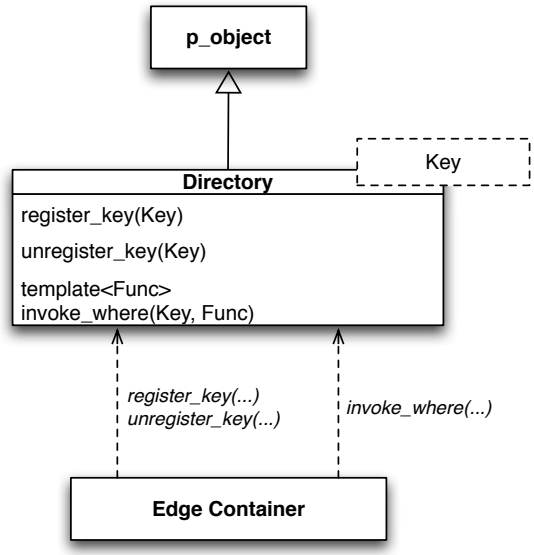


Fig. 33.: The STAPL directory.

responsibility to forward these requests to the appropriate location in the distribution computation.

The task identifier directory is an instance of the general STAPL `directory` class template, which is also used in the `pContainer` framework for a similar purpose. The interface is shown in Figure 33, and is instantiated with a parameter of $task\_id\_t$ for Key. It is a *p_object* which is distributed across the same set of locations as the *graph manager*. As tasks are created, they invoke $register\_key(Key)$. Internally the directory applies a function which maps each key to a location, deterministically choosing a *manager location* where it will store information about this key, namely what location invoked $register\_key()$. Objects wishing to message the location that registered the key can invoke the nonblocking method $where\_invoke(Key, FunctionObject)$. The directory will guarantee the function object is executed on that location by sending

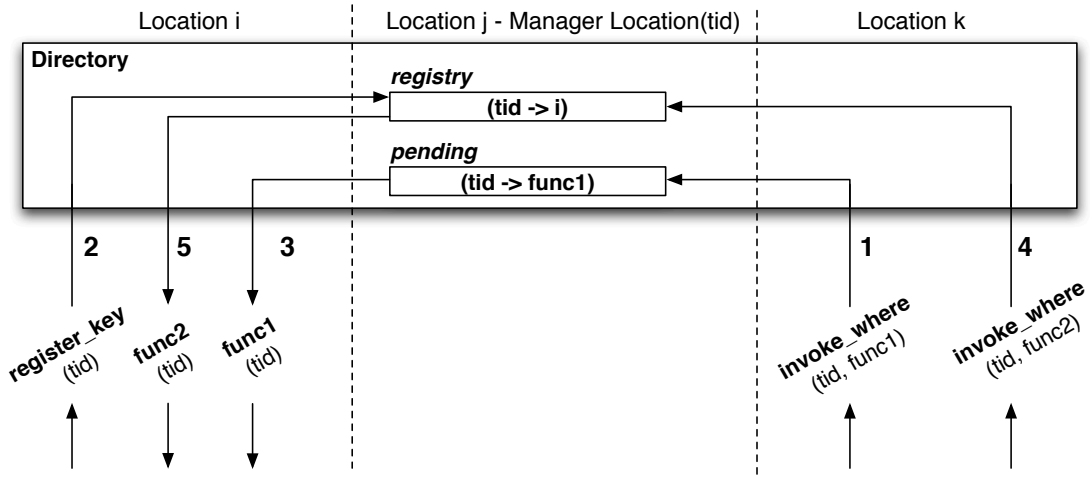


Fig. 34.: Example use of the STAPL directory.

One possible ordering is depicted. Invocation request (1) arrives prior to registration and is buffered in *pending*. Registration occurs (2), location stored in *registry* and buffered request forwarded (3). A second request arrives (4) after registration, and is immediately forwarded (5) to the registering location.

it to the *manager location* which will forward it appropriately. The function object can be of any type; it must only define a function operator that receives Key as a parameter, as the directory will pass it the corresponding value when invoking it on the target location. The approach is similar to that proposed in [37] for Charm++ parallel arrays, though our design is more general and provides stricter guarantees on the ordering of $where\_invoke()$ requests [55].

The important feature of the directory is that no ordering constraints are placed on the $register\_key()$ and $where\_invoke()$ method calls. Function object forwarding requests can be initiated without knowing the status of key registration; requests will be buffered at the manager location until the key registered. This behavior is depicted

in Figure 34. Within the context of the `PARAGRAPH Executor`, this feature allows consumers to request propagation of a task's return value, unconcerned with both *where* its predecessor executes and *when* it is created. On the consumer location, the *edge container* sends such requests via *where_invoke*() and then immediately proceeds with local task creation and execution activities.

As mentioned earlier, the graph manager allocates the identifier directory, and the directory's lifetime is directly tied to the manager. Only after all tasks have completed execution can the framework be sure that no additional tasks will be created, which may require the services of the directory. As we will see in the next section, however, the edge container can detect when a task will receive no further forwarded requests. When it does, it can invoke *unregister_key*() to reclaim memory in the registry stored on the manager location.

V.5. The Edge Container

As discussed in previous sections, the *graph manager* is responsible for initializing the task graph and providing a task specification interface to workfunctions. The `task` objects it creates are passed to the *scheduler* to carry out the computation the user requested, once dependent tasks have finished execution. Ensuring these dependences are satisfied, providing access to the task's return values, and notifying consumer tasks of these events in a distributed and scalable manner is the responsibility of the *edge container*.

The edge container provides random access to the values produced by tasks in the `PARAGRAPH` and is indexed by the *task identifier type* (typically `size_t`). Guided by its domain of application, the container adheres to a *write once, read many* policy of the values it maintains. Each entry's value is set when the producer task completes

(via the `set_edge()` method), and consumers are provided read-only access to it. Given these access constraints, the edge container's default policy is to create copies of the produced value on locations with active consumers (i.e., in the WAIT or EXEC stage). Multiple consumers on a location share the same value, so it need only be sent once, despite multiple edges in the `PARAGRAPH`. This transparent replication is done in an effort to reduce latency during task execution; but the behavior can be trivially overridden by having a workfunction return a view of the element, as opposed to a value. In this case, the replication decision is transferred to the view.

In addition to producer writes and consumer reads during their execution, tasks invoke initialization functions, (`add_producer()` and `add_consumer()` respectively), during their creation (see Section V.3.3). As described in Section IV.2.3, binding edge specification to the task creation process allows us sufficient state to appropriately set the lifetime of produced values and keep consumers from premature execution, while enforcing no ordering on these creation events. Once they both have occurred, sufficient information exists to create the initialize edge in the *edge container*, allowing proper consumer notification and value propagation when the producer has completed execution.

In this section, we describe the implementation of the edge container, beginning with a small illustrative example and then a description of the relevant associated classes in the framework. We continue by describing the four major application events that catalyze change in the container's state. We then discuss the conditions for the removal (or the *eviction*) of edges from the container, describing the conditions that dictate whether a producer task is kept in the HELD stage or transitioned to RETIRED. We conclude by discussing an extension to the basic edge container, named *partial edge consumption*, which allows successors to specify consumption of only part of the return value of a producer task, with the aim of reducing cross-location value

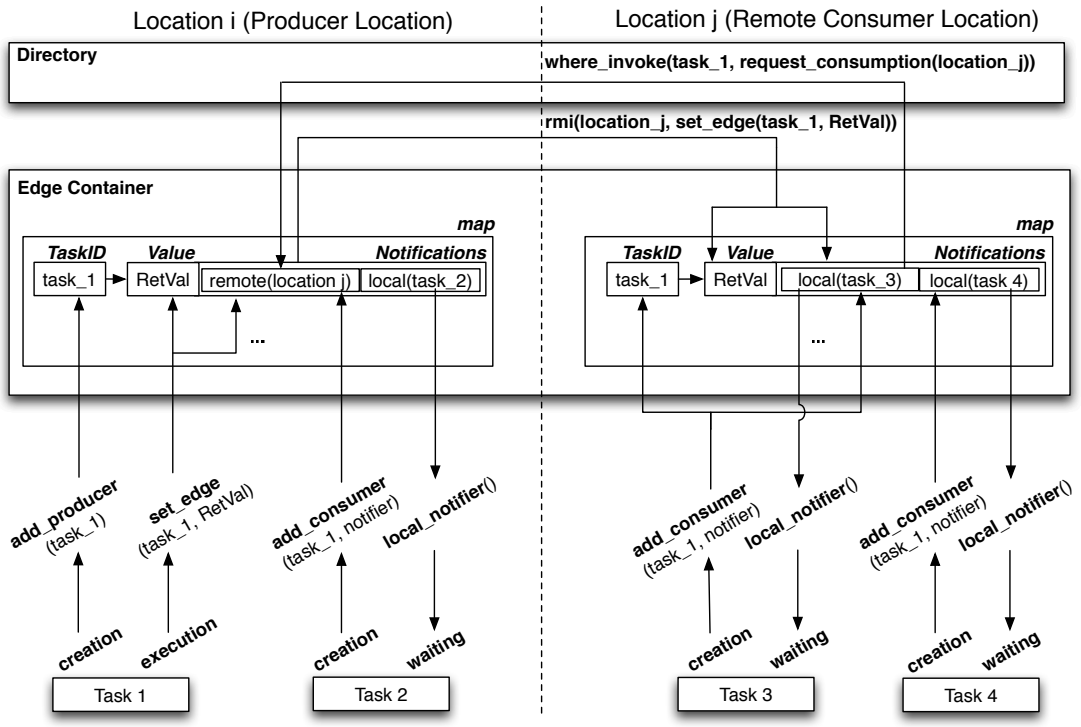


Fig. 35.: Example use of the edge container. One producer with a single local consumer and two remote consumers.

propagation costs. (i.e., required communication bandwidth).

V.5.1. Example Usage

We will soon describe the edge container implementation in detail; however, we begin with an example of edge container use in order to provide some context for this discussion. In Figure 35, we consider a simple single producer, multiple consumer dependence graph. Specifically, $Task1$ has 3 consumers; one of them, $Task2$, is created on the *producer location*, i. Two other successors, $Task3$ and $Task4$, are *remote consumers*, that will execute on location j. In this example we examine

a single ordering of events shown below using the notation previously described in Section IV.2. However, the edge container will appropriately handle any sequence of events respecting the partial ordering defined in Figure 10.

$$T_{C_1},\ T_{C_2},\ T_{C_3},\ T_{C_4},\ T_{E_1},\ T_{E_2},\ T_{E_3},\ T_{E_4}$$

When $Task1$ is created, `add_producer()` is called and an *entry* is created on the producer location in a map data structure, with the task identifier serving as the key. The data associated with this key is storage for the value that will be produced and a list for consumers to register notification requests. Next $Task2$ is created, and searches for an entry in the map for task identifier $task\_1$. Having found one, the local consumer registers itself and proceeds with creating its `task` object, using the value reference created by calling `consume()` (Section IV.3.4), which is in fact backed by the $RetVal$ stored in this map.

Continuing on, $Task3$ is a consumer created on a remote location. It unsuccessfully looks for an entry for $task\_1$ in its local map, and therefore creates one, registering a *local notifier*, similar to $Task2$. However, before returning, it employs the *directory* to forward a request to the producer location. This request causes a *remote notifier* to be registered in the producer location's map entry. To conclude initialization, $Task4$ is created and having found an existing entry, registers its local notifier, but need not request remote notification, as this has previously been done by $Task3$. However, as we will see later, it must still inform the producer location of its creation.

After $Task1$ executes, it invokes `set_edge()` which sets $RetVal$ in the map entry and invokes all notifiers. Locally, $Task2$ is notified that the value is available, satisfying the dependence and releasing it to execute. The producer location also invokes `set_edge()` on the remote consumer location via the remote notifier (which

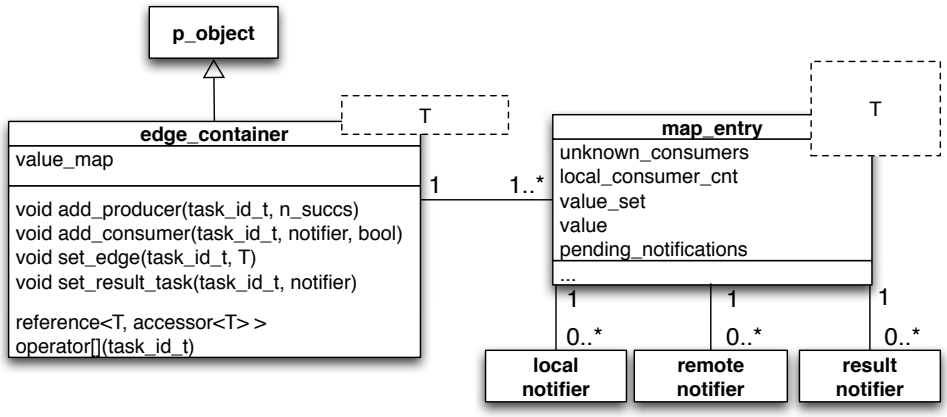


Fig. 36.: The `edge_container` class and related classes.

uses `ARMI`). The value is inserted locally there, and all notifiers are serviced, freeing $Task3$ and $Task4$ to execute. Since all consumers have been notified, the entry on each location is deleted when its local consumers have finished execution and no longer require access to the produced value.

V.5.2. Classes

In Figure 36, we show the edge container and associated classes. The `edge_container` class template is a `p_object` and has one primary data member, the *value_map*, which has entries for all tasks that have been created and not yet retired. The `operator[]()` method is used to implement the `consume()` interface of the dynamic workfunction. Four other public methods correspond to events which will be described in the coming sections.

The `map_entry` class contains all of a location's state associated with a given edge, keyed by the task identifier. *unknown_consumers* tracks the number of successors yet to be created, and *local_consumer_cnt* counts number of local references

outstanding to an edge value. It is incremented during the consumer task creation and decremented when the reference is destructed. *value_set* is *true* if the produced value is available in the *value* field on this location. Finally, *pending_notifications* keeps a list of the callbacks that should be invoked when a value becomes available on a location. We have previously discussed the `local_notifier` class and will examine the `remote_notifier` and `result_notifier` in the sections that follow.

V.5.3. Events

The *edge container* is notified via method invocations of various `PARAGRAPH` events, primarily focused around task *creation* and *execution completion*, but also including *result task identifier specification* by the factory (see Section IV.3.6). These events serve to create edge entries in the container, allow successor tasks to be released for execution, and eventually allow these edges entries to be destroyed when their values are no longer needed. In this section, we describe in detail how the edge container responds to each of these events.

V.5.3.1. Adding a Producer

The edge container method `add_producer(`*tid*`,` *n_succs*`)` is called by the *graph manager* during an `add_task()` invocation if the task being created has successors. Figure 37 illustrates what effects this event has on the edge container. First, *value_map* is searched to see if an entry already exists for *tid*. If one does exist, then at least one consumer on this location has already been created (i.e., $local\_consumer\_cnt > 0$). Note that it is impossible for remote consumers to have registered at this point, as any such consumption requests are buffered in the *directory* at the *manager location* for *tid*, waiting for it to be registered. If no entry exists in the *value_map*, one is constructed (with *local_consumer_cnt* default initialized to 0). We next initialize

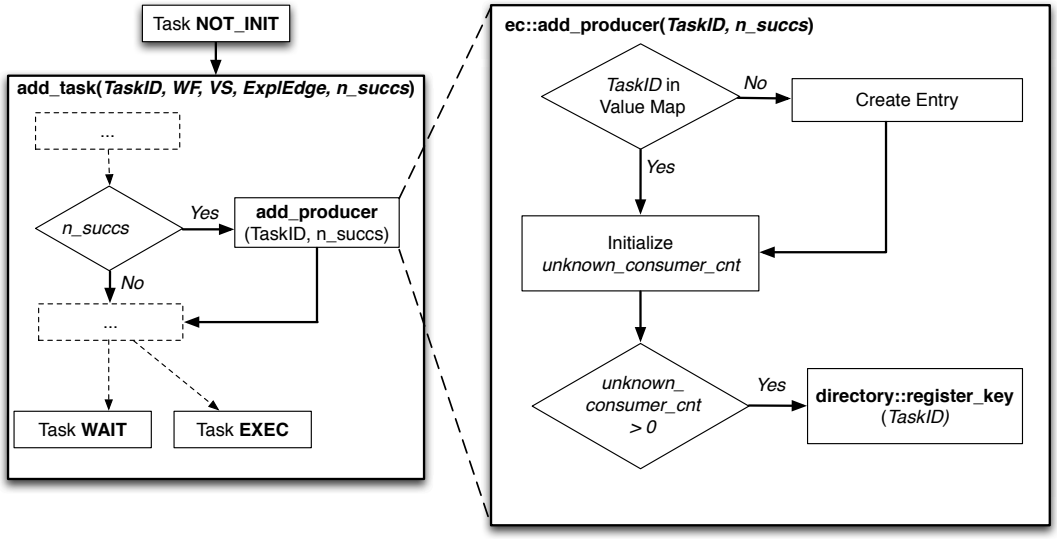


Fig. 37.: The add producer edge container event.

unknown_consumer_cnt as follows:

$$unknown\_consumer\_cnt = n\_succs - local\_consumer\_cnt$$

With this assignment, *unknown_consumer_cnt* now represents the number of consumers that have yet to be created plus those have been remotely created but are buffered (or in flight) to the manager location for *tid*. If this count is nonzero, we call `register_key(`*tid*`)` on the directory so that remote consumption requests can be delivered. However, if this count is zero, then all consumers are local and have already been created. Therefore, we can be certain that no remote locations will need to message us; and we avoid the communication and memory costs associated with registering. With producer-side initialization of the edge the complete, control returns to `add_task()`, where the task will finish its transition from the NOT_INIT state to either WAIT or EXEC, as previously described in Section V.3.3.

V.5.3.2. Adding a Consumer

When the *graph manager* is servicing an `add_task()` request, `add_consumer()` will be invoked for all incoming dependence edges, so that the successor-side initialization of the edge can be completed. This method receives parameters representing the producer task identifier, the successor's `local_notifier`, and a boolean conveying whether this is a value or signal dependence. In Figure 38, we summarize the steps taken in the edge container for each of these invocations. We begin by searching the *value_map* for an entry keyed by the corresponding producer task identifier. If one does exist, then it means that one or both of the following has already occurred on this location:

- **The producer task has been placed on this location and may have finished execution.** We can detect creation by checking if *unknown_consumer_cnt* in the entry is nonzero. This field is only set as such on the producer location and cannot yet have been decremented to zero, as we are a creating a consumer that has not registered with it yet. We can also determine whether this producer has executed by consulting the *value_set* entry field.

- **There is at least one consumer already on this location in the** WAIT **or** EXEC **stage.** If an entry exists and this location is not producing the value, then a previous consumer is either waiting for notification from the producer location or has entered the EXEC stage and is still utilizing the value, keeping it live in the *value_map*. We need not send a duplicate notification request to the producer location, but rather simply let it know of our existence as a known consumer.

With this in mind, we proceed by incrementing the *local_user_cnt* field in the existing entry, if this successor is a value consumer. This will hold the value in the

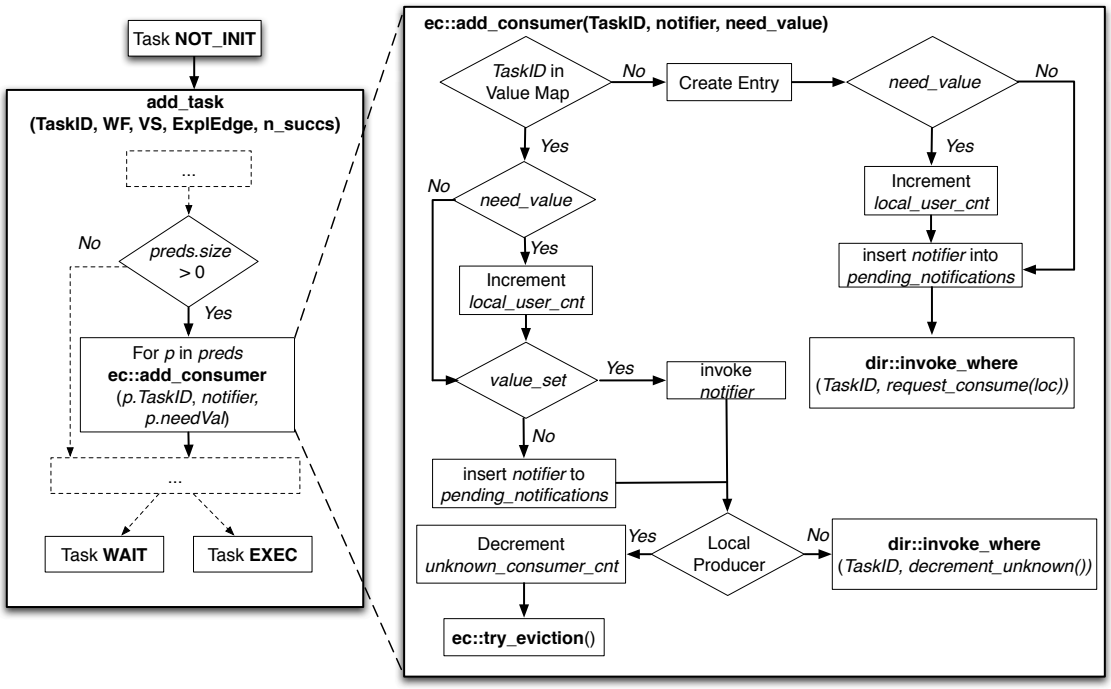


Fig. 38.: The add consumer edge container event.

map even after we are notified, so that task can access it during its EXEC phase. Next *value_set* is checked, and if it is $false$, the notifier parameter is appending to $pending\_notifications$. Otherwise, the notifier is invoked immediately.

Finally, if the producer is local, *unknown_consumer_cnt* is decremented. We attempt entry eviction (see Section V.5.4) at this point to transition the producer task to RETIRED. Eviction will succeed if $unknown\_consumer\_cnt = 0$ and all active local successors had only requesting signaling (i.e., $local\_user\_cnt = 0$). Alternatively, if the producer is remote, we use the directory to forward a request, `decrement_unknown()`, where similar actions will be performed there in the value map entry, namely decrementing *unknown_consumer_cnt* and an entry eviction attempt at that location.

If there is no existing entry, then the producer may execute elsewhere; and the consumer must request the producer location report task completion to this location when complete, as there is no pending consumption request from this location. We create an entry and proceed as before by incrementing *local_user_cnt* if necessary. We then insert the *notifier* parameter to `add_consumer()` in this location's newly created entry's *pending_notifications*. Finally, we employ the directory to forward a `request_consume()` to the producer location, passing it our location identifier. Though not depicted in Figure 38, this request proceeds similarly as if we had found the producer locally with one exception: the notifier the producer location inserts into *pending_notifications* is not a *local_notifier*, but instead a *remote_notifier*, which we will discuss in the next section.

Note that such remote edge setup communication is always initiated from the consumer task's location when operating in *source exact, target count*, which is our current implementation choice. Furthermore, as we will see when discussing `set_edge()` below, notifications from the producer to remote consumers do not need to employ the *task identifier directory*. Therefore, we do not call `register_key()` for consumer tasks that do not themselves have successors, as this would generate needless communication traffic and memory usage in the directory.

In the implementation described here, we always transmit the value from the producer location to all successors, even to locations with no true data consumers (i.e., tasks just requesting notification via `wait()`). This does not cause problems or persistent memory usage, as the value will be immediately evicted after all local notifications finish. However, it does introduce unnecessary communication. Avoiding these transmissions complicates the protocol, requiring us to maintain information about the *type* of previous dependence requests on a given location. Furthermore, we must provide a mechanism to *upgrade* the previous request type. For example, a

data consumer initialized on a location after another successor only desiring a signal, cannot simply send a `decrement_unknown()` to the producer location, as it would now (i.e., where we always send the entire value). Instead, it must inform the producer that this remote location now needs the value to be transmitted. In Section V.5.5, we discuss an extension to our implementation that resolves this inefficiency and provides even more general edge consumption facilities.

There is one interesting point that is worth mentioning: when `add_consumer()` fails to find an existing local entry, while it can be certain that the producer has not been previously locally initialized here, it is possible that previous consumers were initialized, notified, and completed execution, leading to eviction of the produced value from map. The protocol described above, easily handles such cases, issuing a new `request_consume()` request to the producer (which we know will be immediately serviced). However, this leads to additional communication costs that could be avoided by changing the entry eviction policy. We will describe the possible changes when we discuss the eviction policy.

V.5.3.3. Setting an Edge Value

When a task has invoked the user's workfunction and received the return value, the `set_edge()` method of the edge container is called to propagate this value to all locations with consumers and notify these successors of this now available edge. The activities involved in this operation are shown in Figure 39. First, the entry for task identifier is found in the edge container's local *value_map*, where we subsequently set the entry's *value* field to the workfunction's return value and change *value_set* to *true*.

We next flush $pending\_notifications$, invoking all previously registered callbacks ($pending\_notifications$ will henceforth remain empty, as future consumption requests

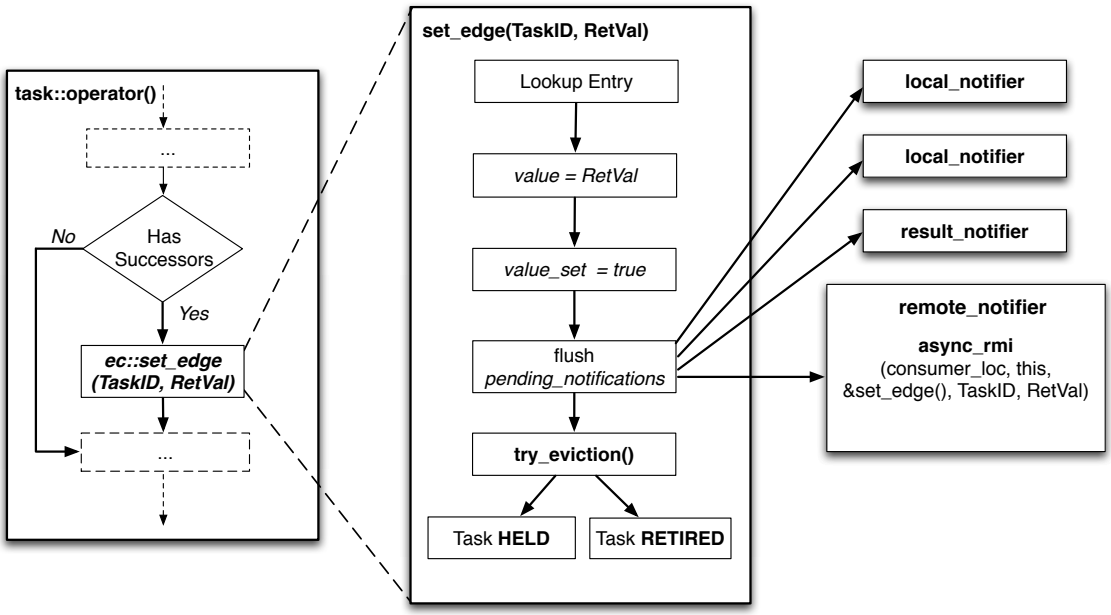


Fig. 39.: The set edge event.

will be immediately notified). This list is a mix of `local_notifier`, `result_notifier`, and `remote_notifier` callbacks. We have discussed the first previously, and will discuss the second in the next section. The behavior of the third is depicted in Figure 39. This remote callback triggers a remote method invocation directed at a consumer location. `ARMI` will invoke `set_edge()` there, where the value will be similarly set and pending $local\_notifier$ callbacks will be invoked. Note that in the current implementation, no remote notifiers exist on these consumer locations, though they could be similarly serviced in a recursive manner. In Chapter VIII, we discuss briefly how this insight suggests that a hierarchical edge container implementation, with aggregated consumption requests and distributed edge value dissemination, should be a useful and straightforward extension of the current protocol.

In both the producer location's invocation of `set_edge()` and those remote invo-

cations it spawns, we conclude the method by attempting eviction of the *value_map* entry at the respective location. On the producer side, part of this process is examining *unknown_consumer_cnt*. If it is nonzero, the task transitions from EXEC to the HELD stage, where it waits to service post-execution consumption requests. When it does reach zero, `unregister_key(`*task_id*`)` is invoked on the directory, as no future request will be made of the task. We then continue the `try_eviction()` process as described in Section V.5.4 on the producer location in the same manner that it is independently applied on each remote consumer location.

V.5.3.4. Setting the Result Task Identifier

When the factory workfunction invokes `set_result_task()` for the purpose of *return value delegation* (see Section IV.3.6), the graph manager forwards this request to the edge container by creating a notifier and passing it along with the specified task identifier to `set_result_task()`. This `result_notifier` is a callback to the storage backing the *return_view* of the `PARAGRAPH` and gives it a chance to copy in the producer value and notify its successors. This invocation of `set_result_task()` subsequently calls:

$$\texttt{add\_consumer}(\textit{task\_id}, \textit{notifier}, \textit{true}),$$

where, as previously discussed, the last boolean parameter specifies that this is an actual data consumer and not just a successor requesting signaling. The notifier is thus added to the appropriate entry in the *value_map*, and will be invoked when the value is locally available. Note that unlike internal value successor tasks, the value is not guaranteed to remain persistent in the value map after notification. The `result_notifier` must facilitate copying the value, so that the `PARAGRAPH Executor` can proceed with reclamation of this predecessor `PARAGRAPH`'s resources.

V.5.4. Entry Eviction

One of the key goals of the `PARAGRAPH Executor` is to enable incremental execution, not requiring the entire dependence graph to be fully initialized at any one time. Specifically, we wish to make the memory requirements proportional to the *execution window* of the task graph, whose size is defined by tasks that have been initialized (i.e., WAIT) but have not yet completed and entered the RETIRED stage. It should now be evident that prior to creation, a task consumes no such resources, save whatever is encoded in the factory workfunction to specify its initialization. Furthermore, we have also covered how the `local_notifier` and `task` objects are freed, at the beginning and end of of the EXEC phase, respectively.

What remains to be reclaimed is the various *value_map* entries associated with a given task's produced value. These entries exist both on the task's execution location and on all successor task locations. In order for the producer to completely transition to the RETIRED stage, where no system resources remain in use by the task, each of these entries must be evicted using a policy we now describe. As seen in the discussion of the edge container events, there is a small set of task graphs events where the state of a given location's entry may change to an *evictable* state. At each of these we invoke `try_eviction()` which is defined as follows:

$$if\ (evictable(task\_id))$$
$$value\_map.delete(task\_id)$$

`evictable()` returns true if and only if all of the following conditions are met:

$$unknown\_consumer\_cnt = 0$$
$$local\_user\_cnt = 0$$
$$pending\_notifications.size() = 0$$

This straightforward definition is what one would expect. As said before, on the producer location, eviction cannot occur until after all consumer locations are known; on all other locations with entries for the task, *unknown_consumer_cnt* is always zero. In addition, all local users of the value must be finished accessing it. This includes all references used by consumer tasks (i.e., they must at least have progressed to the HELD stage). Finally, all callback notifications must have been serviced, whether they be *local*, *remote*, or *result* notifiers. Note that we do not need to check *value_set* (i.e., that `set_edge()` has been called on the entry's location), as this is implicitly checked by the above conditions. On the producer location, *unknown_consumer_cnt* and $pending\_notifications.size()$ cannot both be zero prior to *value_set* being set. On remote consumer locations, $pending\_notifications.size()$ will always be nonzero prior to this point in any initialized map entry.

Immediate deletion of evictable entries on consumer locations helps ensure a small memory footprint for the task graph's execution window. However, as previously mentioned in the discussion of `add_consumer()`, this aggressive stance may cause additional communication that could otherwise be avoided. Recall that when a new consumer finds an existing entry for task a it wishes to consume, it forwards a `decrement_unknown()` invocation as opposed to `request_consume()` through the directory, avoiding duplicate transmission of the value. Therefore, by delaying the eviction process, we would enable a greater number of future consumption requests to reuse the value, better amortizing the associated communication cost.

Intelligent, delayed entry eviction (i.e., not negating the benefits of incremental generation) requires information not currently available to the edge container. First, knowledge from the STAPL runtime system about the presence of excess memory for such activities is needed. Additionally, interfaces to request reduction in this value caching when necessary need to be developed.

While we could look to previous work in cache policies (e.g., LRU [30]) or edge container generated metrics (e.g., evicting the task identifier with the least number of previous consumers), it is likely that the `PARAGRAPH` developer has some knowledge of the number of consumers for a task on each location. Therefore, in the future we may look to extend the factory interface, allowing the optional specification of such contextual information that could guide better memory reclamation in the `PARAGRAPH Executor`.

V.5.5. Partial Edge Consumption

As previously discussed, it is important to provide an efficient mechanism to specify consumers that do not need the entire value provided by the producer, whether they simply desire a signal from the producer (as their data is located outside the purview of the `PARAGRAPH Executor` in `pContainers`), or they alternatively only need a subset of the produced value, desiring it be *filtered* prior to passing it to them as an input parameter. As described in Table IV, `wait()` exists for the first case; here we show how its implementation can be made more efficient. For the second, we provide a new, optional parameter to the `consume()` function, allowing the workfunction writer to specify a *filtering function* that will be applied to the requested producer's value. We show an example use of this interface in Figure 40, where we assume a predecessor returns a view over a sequence of values, and the successor's `add_task()` invocation specifies that it need only consume the first element in the sequence.

We accomplish this refinement to `consume()` by introducing the notion of a *request level* that can have three possible values: *Signal*, *Partial*, and *Full*. All are unified under the application of filtering functions. *Signals* employ a function returning void, and *Full* is an identify function which returns its input. *Partial* functions represent those passed explicitly by the user to `consume()`.

```
1  // filter function object
2  struct select_first
3  {
4    template<typename View>
5    auto operator()(View view) const
6    {
7      return view[0];
8    }
9  };
10 ...
11 // consumer task creation
12 add_task(wf, n_succs, consume(pred_tid, select_first()));
```

Fig. 40.: Example of partial edge consumption.

Generally, we attempt to apply a remote consumer's filter at the producer's location. Passed by the consumption request and stored in the associated remote notifier, evaluating it here minimizes transmission costs by only transferring those parts of the return value the consumer is interested in. However, in the presence of multiple consumers on the same remote location, it may prove more efficient to apply it on that location after receiving the remote notification but prior to notifying consumers. For example, if a location has one *Partial* consumer and another *Full* consumer, it is better to transmit the full value once and locally apply the *Partial* function, rather than requiring two distinct value transfers from the producer's location.

In the edge container, value map entries track the highest outstanding level of request at a consumer location. When a new `add_consumer()` is invoked by the task, we consult this stored level. If the entry's current request level *covers* the new consumer's request, we proceed as in the basic protocol, forwarding a `decrement_unknown()` call to the producer via the directory.

If a new request is *uncovered*, we send an `upgrade_request()` invocation to the producer location. If it arrives prior to the previous request level for the consumer location being processed, it supersedes this old one, and all requests at the remote location will be serviced by this a new, upgraded remote notifier. If the old notifier has already been serviced (i.e., the producer has finished), we immediately service this second request. The remote consumer map entry maintains sufficient state to progressively invoke its local notifiers based on the request level of remote notifications it receives (i.e., notifiers are paired with the filtering function object associated with each of them).

Determining the request coverage relationship of two *Partial* consumers cannot be accomplished without additional information from the filter functions. Hence, in the current implementation, two outstanding partial consumers on the same location cause the request level to be immediately upgraded to *Full*; and the filters will be applied when the value arrives on that location. However, we believe that the *synthesis* of these partial requests into a new request, representing the union of the required parts of producer's value, is an avenue that warrants further investigation. The filtering functions passed to `consume()` are typically derived from `pView` operations where some high level information exists that can be expressed to the `PARAGRAPH Executor` by the `PARAGRAPH` enabling the efficient coalescing of consumption requests.

V.6. The Scheduler

The STAPL runtime system's scheduler is responsible for task execution and managing the lifetime of the *graph manager* and its related classes. The scheduler is hierarchical; the *root scheduler* associated with `stapl_main` has a list of schedulable *entries*. These entries are either *scheduler entries*, representing additional scheduler instances

associated with nested `PARAGRAPHs`, or *task* entries, which represent `task` objects. It also has a customizable *scheduler*, responsible for ordering the runnable tasks for execution, and an *idle queue* for inactive nested schedulers that have not yet completed. Finally it has a *termination detection* object, the implementation of which we defer for now, but will describe shortly.

In Algorithm V.6.1 we summarize the executor's progression function. Currently, it is invoked when the size of the application's composed `PARAGRAPH` reaches a maximum window size, currently a user defined parameter. It is continually invoked until it has reduced the current execution window by an number of tasks defined as another tunable parameter. It continually requests a task from the scheduler, executing it and checking the status returned by the entry. If it is a scheduler entry, this algorithm is recursively invoked on it, with the child selecting a single task according to its own scheduling policy to execute. Otherwise, it is a task entry, which will return *Finished* and be destroyed, unless it is an incremental task which has signaled it should be re-executed. In that case it remains *Active* and is reinserted into the scheduler for successive invocation.

Nested schedulers can also be *Idle*, denoting that while there is currently no local work to do, there is a possibility that `ARMI` initiated `add_task()` requests will create new local tasks, as the associated task graph has not finished completion. *Idle* entries are placed in a separate idle queue. They remain there until such remote requests arrive and place them back into their parent's scheduler queue, or termination detection succeeds on them, in which case both they and the associated *graph manager* are destroyed.

At each level of the scheduler hierarchy, the next step is to check the scheduler. If it is non-empty, the algorithm returns *Active*. Otherwise, termination detection is invoked on idle children as well as the current scheduler. If all succeed, the current

Algorithm V.6.1 Scheduler progression.

```
1: entry = scheduler.pop()                                  ▷ Get Next Runnable Entry
2: child_status = entry.execute()                                    ▷ Execute Entry
3: switch child_status                          ▷ Return Value Based Post-Processing
4:     case Active : scheduler.push(entry)
5:     case Finished : destroy(entry)
6:     case Idle : idle.push(entry)
7: end switch
8: if !scheduler.empty() then                                ▷ Compute My Return Value
9:     return Active
10: end if
11: done = true;
12: for i = 1 .. idle.size() do                              ▷ Try to Retire Idle Children
13:     entry = idle.pop()
14:     entry_done = entry.termination_detection.make_progress()
15:     done = done & entry_done;
16:     if !entry_done then
17:         idle.push(entry)
18:     end if
19: end for
20: if done & termination_detection.make_progress() then
21:     return Finished
22: end if
23: is_idle = true
24: return Idle
```

scheduler is *Finished*. Otherwise it is *Idle* and will be placed in its parent's idle queue.

V.6.1. Termination Detection

Prior to releasing a task graph's resources on each location (i.e., invoking the graph manager's destructor and releasing other objects such as the edge container and task identifier directory), the scheduler must ensure that all tasks in the distributed computation have completed. Simply emptying the scheduler's current, local pool of assigned tasks for the graph is an insufficient condition to check; tasks with dynamic workfunctions, including the factory, executing elsewhere in the graph may create additional tasks that could easily be migrated to this location. To ensure that each location's graph manager instance remains persistent until this condition is met, the executor employs a *distributed termination detection* algorithm.

A termination algorithm already exists in the runtime system, as it is required for implementing the `ARMI` collective synchronization primitive `rmi_fence()`. Derived from the approach presented in [49], this algorithm employs a series of nonblocking *all_reduce* operations (i.e., a global reduction followed by a broadcast of the value), to guarantee that all remote method invocations requested in STAPL have been received and processed at their target locations. Each location contributes a local *termination condition value* to the global reduction:

$$rmi\_invoked\_cnt - rmi\_serviced\_cnt,$$

where *rmi_invoked_cnt* is the number of RMIs initiated from a location, while the number of RMIs that have been processed there is tracked by *rmi_serviced_cnt*. When the global sum of this local differences is zero, `rmi_fence()` can safely return on each location, as all outstanding RMIs have been serviced. This method is non-blocking, allowing `ARMI` to alternate between servicing arriving RMI request (analogous to our

processing of incoming `add_task()` requests) and making progress on the required sequence of all_reduce operations.

The *termination condition value* for `PARAGRAPH` quiescence is quite similar in form to that used in `rmi_fence()`, involving *graph manger* variables as follows:

$$tasks\_added\_cnt - tasks\_processed\_cnt$$

When all tasks that have been created have been processed, a task graph can safely terminate. With this is mind, a task graph's scheduler constructs a termination detection object, passing it (1) a callback function to the graph manager method that computes the current local termination value and (2) the global value that should be attained for quiescence (sum is currently assumed to be the reduction operator). Following this initialization, forward progress in the termination detection process proceeds as previously described in the scheduler's workflow.

Note that this approach is general, working in all cases; however, many common task graphs can specify a simpler and more precise termination condition, reducing or removing the need for global collaboration to determine that no further tasks will be executed on a given location. For example, the `map_reduce` pattern ends with a broadcast of the reduction variable to all locations in the `PARAGRAPH`. Reception of this value can sufficiently serve as termination notification. To allow this important optimization in these cases, the termination detection class template can be customized by the `PARAGRAPH` via partial template specialization. Specializations must simply accept a callback function object to invoke when they detect termination. In turn, they typically register notifiers with the edge container (through additional graph manager interfaces) with one or more task identifiers that together represent the last tasks that will be executed on each location.

CHAPTER VI

OPTIMIZATIONS FOR THE PARAGRAPH EXECUTOR

In this chapter, we consider two techniques the `PARAGRAPH Executor` employs to decrease the cost of accessing elements referred to by views. To maximize data locality, *task placement* decouples the location where a task is inserted into the graph (via `add_task()`) from the location where it will be executed, collaborating with the task's workfunction and views in the placement decision. *View localization* further optimizes data access after task placement has finished. When the elements an input view references are confined within the location where the task executes, view localization removes the intrinsic overhead in the `pContainer` address translation mechanisms, which are necessary to support a *shared object view* in systems with distributed memory.

VI.1. Task Placement

In the `PARAGRAPH Executor`, we choose to make the decision of where a task will execute independent of the location that creates it to maximize performance. There are two reasons this task migration in STAPL may reduce application execution time. First, the initiating location may already have excess tasks to process; and moving the execution of the new task to a less loaded processor would better balance the system's computational load. Second, migrating a task to an alternate location may increase locality by placing it where `pView` element accesses incur less latency. In this dissertation, we focus on a task migration policy which addresses this latter concern.

In this section, we begin by describing how the `PARAGRAPH Executor` interacts with the views of a task to implement a basic task placement policy. We next describe two refinements to the basic policy, one that takes into account how the views will be

```
1  enum location_qualifier
2    {LQ_CERTAIN, LQ_DONTCARE, LQ_LOOKUP};
3
4  class LocalityAwareView
5  {
6    // serialization required for migration
7    void define_type(typer& t);
8
9    pair<stapl_location , location_qualifier>
10   preferred_location(void);
11 };
```

Fig. 41.: Required view operations for task placement.

accessed in the workfunction and another that efficiently implements task placement when views have locality information distributed across the system.

VI.1.1. View Interfaces and the Basic Placement Policy

In Figure 41, we present the interface that the task graph requires of a view to support migration. Currently, all views in STAPL are required to provide this interface, though most inherit the functionality from base view classes that provide such core functionality. Though highly recommended, views are not strictly required to provide this interface; views passed to tasks that do not provide it are simply not consulted in the task placement decision.

Two methods must be defined by the view for task placement. First, the view object must describe data marshaling, so that it can be moved from one location's address space to another if it is indeed migrated. This serialization functionality is achieved by implementing the `define_type()` function, which is used by the `ARMI` runtime system component [57].

The second method, `preferred_location()`, returns a pair which describes the locality of the view. The first element of the pair is a location identifier. The second element is a qualifier from an enumerated data type, `location_qualifier`, which is used to choose how the returned location is interpreted and used by the placement policy. In the basic policy, only `LQ_CERTAIN` and `LQ_DONTCARE` are used. As their names imply, the former states that the view is certain that locality for its referenced elements is maximized at the specified location, while the latter states that the view does not have a location preference and that the location element of the pair should not be considered in the decision process. An example of a view that may return `LQ_DONTCARE` qualifier is a view whose data may be distributed throughout the system. In this case, some remote data access will be incurred regardless of task placement. Another example are views backed by the *edge container*. Creation of the edge is deferred until after migration, with the predecessor task's flowed value directed to whichever execution location is chosen by the policy. Usage of the third possible location qualifier value, `LQ_LOOKUP`, is described in SectionVI.1.3 where we extend the basic policy.

Algorithm VI.1.1 shows the basic task placement policy. Note, that the return value of the algorithm is the same as `preferred_location()`. Briefly stated, a location election vote is taken among views returning an LQ_CERTAIN qualifier. The location receiving a plurality of votes is chosen. Ties are broken by preferring the votes of views appearing earlier in the task workfunction's parameter sequence. The reasoning for this heuristic is that many algorithms adapted from the C++ standard library place views read from first, followed by views written to. As we discuss in the next section, there is reason to give preference in the policy to views that are read over those that are written. If there are no votes (i.e., all views return LQ_DONTCARE), then the task is left to execute at the initial location.

Algorithm VI.1.1 Basic task placement.

```
1: procedure BasicExecutionLocation(v_1, v_2, ...v_n)
2:     for i ∈ (1...n) do
3:         vote_i ← preferred_location(v_i)          ▷ Retrieve pair specifier from view
4:         if vote_i.second ≠ LQ_DONTCARE then
5:             increment(vote_tally[vote_i.first])   ▷ Increment location's vote count
6:         end if
7:     end for
8:     if vote_tally ≡ ∅ then                        ▷ No view returned LQ_CERTAIN
9:         return pair(this_location, LQ_DONTCARE)
10:    end if
11:    for location ∈ vote_tally do                  ▷ Create location set for tiebreaker
12:        if location ≡ max(vote_tally) then
13:            max_set.insert(location)
14:        end if
15:    end for
16:    for i ∈ (1...n) do                            ▷ Break ties with parameter order
17:        if vote_i.first ∈ max_set then
18:            return pair(vote_i.first, LQ_CERTAIN)
19:        end if
20:    end for
21: end procedure
```

VI.1.2. View Access Specifiers

We can improve upon the basic placement policy by leveraging additional information about how a task's workfunction will use the views. Specifically, we consider whether a view is read and/or written in the decision process, placing priority on the votes of views that are read. When accessing non-local memory in a distributed system, reads require synchronous communication (i.e., issue a fetch request and wait for the result). In contrast, remote writes can be issued asynchronously, allowing local computation to proceed without blocking.

Algorithm VI.1.2 Access specifier based task placement.

1: **procedure** AccessExecutionLocation($wf, v_1, v_2, ...v_n$)
2: $ReadViewSet \leftarrow \forall\ v_i \in (v_1, v_2, ...v_n)$ s.t. access$(wf, v_i) \in (R, RW)$
3: $pair \leftarrow$ BasicExecutionLocation($ReadViewSet$) ▷ Read View Election
4: **if** $pair.second \equiv LQ\_CERTAIN$ **then**
5: **return** $pair$
6: **end if**
7: $WriteViewSet \leftarrow (v_1, v_2, ...v_n) - ReadViewSet$
8: **return** BasicExecutionLocation($WriteViewSet$) ▷ WriteView Election
9: **end procedure**

Algorithm VI.1.1 shows the modified task placement policy that considers view access specifiers, as discussed in Section IV.3.7. First a location election is held among views which may read data (i.e., Read or ReadWrite). If no read views exist or if they

all return an LQ_DONTCARE qualifier, a second election is held among the write-only views to determine where the task will be executed.

VI.1.3. Forwarding

It is possible that an input view cannot answer the preferred_location() query without performing communication. For more complex containers, such as pGraphs, a directory based method, similar to that presented in Section V.4, is used to divide this distribution metadata throughout the system. Hence, without care, the preferred_location() call could generate blocking, synchronous communication, something we strongly work to avoid. To accomplish asynchronous behavior, we *forward* calls to add_task() that are in the process of the election to locations where this metadata is stored, so they can gather the appropriate information to make the placement decision.

Views that require this forwarding support, notify our framework by returning the LQ_LOOKUP location qualifier paired with the location which will have the distribution information they require. execution_location() returns this qualifier and location identifier, and the add_task() call is transferred to there, where the election will be rerun, allowing the view access to the information it needs. Internally, the view caches this information, as the location is likely just an intermediate stop on the task's journey to its execution location, if the election moves it elsewhere. Furthermore, multiple input views may require additional forwarding to resolve their preferred_location() method to an LQ_CERTAIN state.

The changes to the previous task placement algorithm are straightforward. We continue forwarding the task, rerunning election until all views returning LQ_LOOKUP are able to definitively answer the preferred_location() query. Once we forward through sufficient locations such that all input views return LQ_CERTAIN, the place-

ment algorithm returns a location with an `LQ_CERTAIN` qualifier, by employing the standard placement policy. The task is then placed on this location, where it is initialized as discussed in Section V.3.3.

VI.1.4. Customizing the Placement Policy

As is evident from the previous sections, task placement can be influenced by a variety of factors. The increasingly complexity suggests that any general policy will fail to adequately serve all combinations of algorithms and data distributions. Our default implementation is a reasonable approach and could likely be refined further to broaden its applicability. However, there will inevitably be times that it is best to customize the placement policy for a specific algorithm or view of the data.

An example where such enforced behavior proves useful is in the STAPL matrix algorithm collection. For algorithms such as matrix multiplication, STAPL attempts to use vendor supplied BLAS [38] implementations whenever possible (see Algorithm VI.1.3), as these highly tuned libraries usually provide substantial performance gains. However, there are data layout conformability requirements to use such libraries. Parallel BLAS implementations [16, 23] support a restricted set of data distributions across the machine and internally handle interprocessor communication. If these constraints are met by the problem instance (or if STAPL chooses to transform the problem to a form that does), `p_matrix_multiply()` invokes PBLAS by wrapping it in a STAPL `PARAGRAPH`.

However, if PBLAS cannot be invoked, STAPL employs a general approach outlined in [12], that decomposes the problem into a series of smaller, sequential matrix operations. Each of these operations is a task in the `PARAGRAPH` that will be subject to the task placement policy. Significant performance improvements can still be attained by invoking sequential BLAS routines. These routines are understandably

Algorithm VI.1.3 STAPL matrix multiplication.

```
1: procedure P_MATRIX_MULTIPLY((A, B, C))
2:     if input conforms to PBLAS then
3:         call PBLAS
4:     else
5:         decision =  redistribute | general
6:         if decision == redistribute then
7:             p_copy A, B, C to temporary PBLAS conformable storage
8:             call PBLAS
9:             p_copy temporary result to C
10:        else
11:            call general_matrix_multiply (use BLAS in sequential sections if possible)
12:        end if
13:    end if
14: end procedure
```

```
1  template<typename WF, typename ViewSet>
2  class task_placement_policy
3  {...}; //default policy
4
5  template<typename ViewA, typename ViewB, typename ViewC>
6  class task_placement_policy<
7    general_matrix_multiply_wf,
8    tuple<ViewA, ViewB, ViewC> >
9  {
10 public:
11   void operator()(ViewA, ViewB, ViewC c)
12   {
13     return c.preferred_location();
14   }
15 };
```

Fig. 42.: BLAS customized task placement policy.

unaware of the distributed environment in which STAPL is running and require raw pointers to the input and output matrices involved in the computation. In order to meet this requirement, it is necessary to place the task where the output sub-matrix is located. While the read-only input views can safely copy elements to this location, the written output view cannot be replicated to another location, as this would cause memory coherency issues that are not easily resolved. Note that this preference of write view location over reads runs contrary to the behavior of our default placement policy. Therefore, it is necessary to customize the policy for the `general_matrix_multiply()` algorithm.

The source code for the customized STAPL matrix multiplication task placement is shown in Figure 42. The placement policy is a class template that can be partially specialized on either the workfunction or view types. In this case, we specialize the template such that any tasks created with `general_matrix_multiply_wf` as the

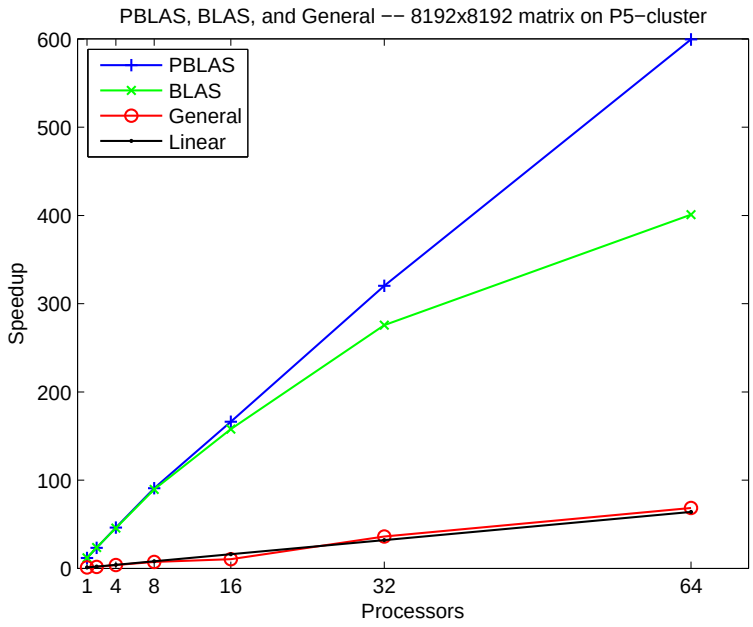


Fig. 43.: Matrix multiply P5-CLUSTER with 8192×8192 matrices. Data type is double. Speedups of the unspecialized (General) and BLAS specialized (BLAS) versions of the algorithm using customized task placement.

workfunction query their third view for its preferred execution location, return this and the associated location qualifier.

In Figure 43, we show the speedup of the BLAS optimized STAPL algorithm over the non optimized version (general), in addition to the STAPL PBLAS version. We ran this experiment on a 832 processor IBM RS/6000 with dual Power5 processors available at Texas A&M University (called P5-CLUSTER) using a matrices of size 8192×8192. Note the dramatic speedups possible using BLAS, whose use is enabled by customizing the STAPL task placement policy.

VI.2. View Localization

Views typically refer to elements in `pContainers`, employing a domain of indices that are *mapped* to those of the container. Users can manipulate these elements in a uniform manner, unconcerned with the reality that these elements may in fact be scattered across multiple address spaces in the distributed computation.

However, even if a task is placed on a location where one or more of its views refer to only local elements, they will conservatively employ the distributed address resolution mechanism of the `pContainer`, checking each access request to see whether it refers to local or remote elements. This translation is understandably more expensive than standard, sequential container retrieval; after determining the element accessed is local, the `pContainer` must then map the index to the correct `bContainer` storage component (as discussed in Section III.1). In the `pContainers` we have implemented, this leads to access times that are greater than that of a sequential, local container. To avoid this overhead, the `PARAGRAPH Executor` explicitly *localizes* view accesses, performing this test once per task instead of for every access, ensuring that the domain of indices it maps to in the `pContainer` is confined to the execution location. In the general case, this test must be done at runtime, to account both for dynamic data distributions and as well as the effects of *task placement*.

View localization occurs during task creation (see Section V.3.3) and gives the view the opportunity to transform itself, constructing a new view, usually of a different concrete type, but adhering to the same `pView` concept (e.g., 1Dview). The new view is defined directly over one or more of the `pContainer`'s local `bContainers`, removing the address translation layer of indirection. The *graph manager* attempts localization for each view input to a task by querying the view's `localizable()` method, defined in Figure 44. If the return value is true, the `PARAGRAPH Executor` constructs a new

```
1 class LocalizableView
2 {
3   // type transformation applied during task creation.
4   typedef ... localized_type;
5
6   // returns true if elements confined within a location.
7   bool localizable(void);
8 };
```

Fig. 44.: Required view operations for view localization.

view of type `localized_type` with the original passed as a parameter. This view is then substituted for the original when the task is executed. Importantly, note that since the workfunction receives views whose types are template parameters, this optimization is completely orthogonal to the workfunction's implementation and occurs transparently at runtime.

VI.2.1. View Dependent Workfunction Return Types

Caution must be taken when attempting to apply localization to tasks with workfunctions that have *view dependent return types*. Localizing a view changes the type passed as template parameter to the workfunction. Hence, if the function's return type is computed using this transformed view type, it will be different than if localization had not taken place.

Consider the workfunction presented in Figure 45. The STAPL `find()` algorithm is invoked, which employs the `map_reduce` primitives to return a reference to the first element equal to 5 in the input view. If localization succeeded, however, this is a reference based on a localized view backed by the a `pContainer`'s `bContainer`, as opposed to the more general `pContainer` backed reference. External consumers

```
1  class view_dependent_result_wf
2  {
3  public:
4    template<typename View>
5    typename View::reference
6    operator()(View view)
7    {
8      return stapl::find(view, 5);
9    }
10 };
```

Fig. 45.: Workfunction with a view dependent return type.

of this task, however, cannot be assumed to have this same localization guarantee; they very likely are executed on a different location, where this localized reference would not be valid. Therefore, to validly apply the view localization optimization, we must be able to subsequently *generalize* any return value that has inherited this localization trait.

We can detect such cases statically, employing a type metafunction [1] to pinpoint workfunctions where varying the view template parameter causes a return type change. If a change is detected, we investigate further at compile time to determine if it is of a form we can currently recognize and properly generalize. If so, we change the task's workfunction type, *wrapping* the user specified workfunction with a workfunction we generate that will intercept it's return value and apply the appropriate generalization transformation. If we do not support this generalization, we conservatively disable any runtime attempts at localization. In this manner, we isolate the scope of localization within the task. Successor tasks receive the return value uniformly, unconcerned with whether any specific optimization were employed on the

producer. They will independently attempt another localization optimization within the locality context of their execution.

Currently, we only support generalization for workfunctions with one view input. Furthermore, the return type should be either the same as the passed view parameter (i.e., the function receives `view_t` and returns `view_t`) or a nested reference type (as shown in Figure 45). In these cases, we invoke the constructor of the *general* return type, passing it the *localized* return value as a parameter. The current level of support is not indicative of a fundamental limitation of the approach, but rather represents cases we have encountered so far. Again, all of the optimization effort is transparent to the end user; their code will remain unchanged and run, regardless of whether the optimization is applied. In the future, we will continue to expand return value generalization support as needed.

CHAPTER VII

EVALUATION

In this chapter we evaluate the performance of STAPL algorithms and applications using the `PARAGRAPH Executor`. We first consider a synthetic benchmark to investigate incremental generation and overlapped execution. We then look at generic algorithms similar in form to those found in STL. Next, we show results for benchmark kernels from the NAS [7] parallel benchmark suite. Finally, we demonstrate the use of STAPL in full applications, presenting one we have developed from the domain of nuclear engineering.

VII.1. Experimental Setup

We conducted our experimental studies on three parallel machines with different processor architectures and network interconnects. These machines include a 38,288 core Cray XT4 (CRAY4) [43] and a 153,216 core Cray XE6 (CRAY 6) [44], both of which are available at NERSC. We also employed a 832 core Power5 Cluster (P5-CLUSTER) [56] available at Texas A&M University. The CRAY4 has 9,572 compute nodes each with a quad core Opteron running at 2.3 GHz and a total of 8 GB of memory (2 GB of memory per core). The compute nodes are connected to a dedicated SeaStar2 router through Hypertransport with a 3D torus topology which ensures low-latency, high bandwidth communication. The CRAY 6 has 6,384 compute nodes each with two twelve-core AMD 'MagnyCours' running at 2.3GHz and total of 32GB of memory. These compute nodes are also connected with a 3D torus via Cray's 'Gemini' interconnect. The P5-CLUSTER is a 832 processor IBM cluster with p575 SMP nodes and 16 cores per node. In all experiments, a location contains a single processor core, and the terms can be used interchangeably.

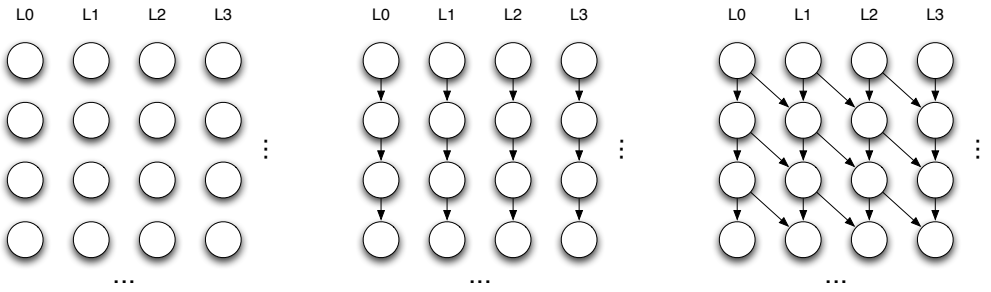


Fig. 46.: Synthetic patterns to study incremental generation.
(nodeps) Each location has a set of independent task to execute. (localdeps) Each location has a serial sequence of dependent tasks. (remotedeps) Tasks have one local as well as one remote predecessor and successor.

VII.2. Incremental Generation

We investigated the overhead of incremental generation in the `PARAGRAPH Executor` as well as the overlapped creation and execution model that enables it. We consider three simple, synthetic task graph patterns which are depicted in Figure 46. All three patterns construct a series of tasks concurrently on multiple locations that can be visualized as a two dimensional matrix of tasks. The *nodeps* pattern has no edges in the graph. The *localdeps* pattern has a serial chain of dependences down each column (i.e., location). Finally, the *remotedeps* graph has an additional diagonal dependence across columns, representing edges that require the `PARAGRAPH Executor` to employ the facilities discussed in Chapter V.

We perform the experiment on P5-CLUSTER, fixing the columns (i.e., processor count) at 128 and the number rows (i.e., tasks per processor) at 1000. Each task

employs the `nano_sleep()` system call to simulate a workload taking one millisecond. We vary the number of incremental task factory calls the `PARAGRAPH` uses to populate the graph from 1 to 100 to study the affects of the approach on execution time. For each factory call count and dependence pattern pair, 30 iterations were executed to create the associated 95% confidence intervals for the results.

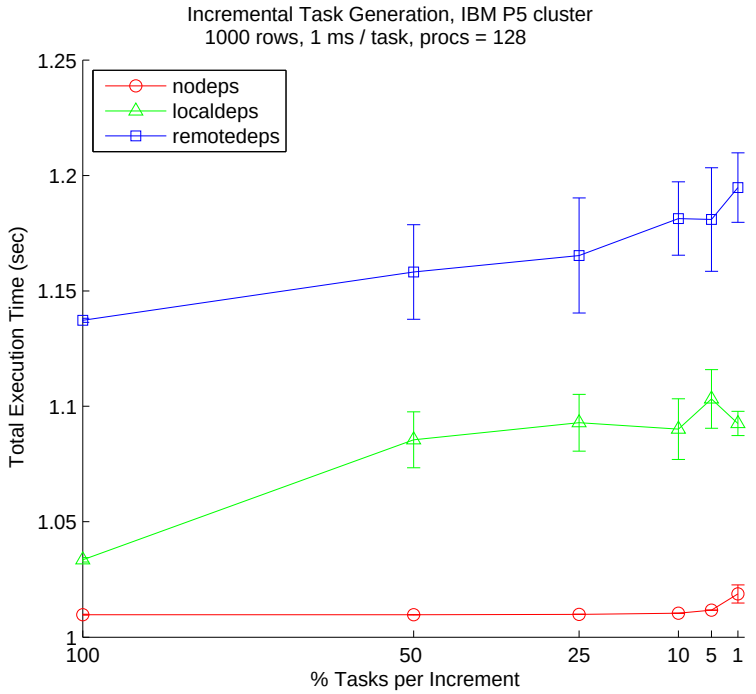


Fig. 47.: Incremental generation test on P5-CLUSTER. 95% confidence interval.

The results of the experiment are shown in Figure 47. The *nodeps* pattern gives insight into the minimal overhead the `PARAGRAPH Executor` will incur when creating and managing tasks. The sum of time spent in workfunction is one second (1K

instances, each taking 1ms). The total overhead of the `PARAGRAPH Executor` in this case is approximately 1% for all cases. The *localdeps* pattern understandably has more overhead, as the `PARAGRAPH Executor` must now track and enforce dependences.

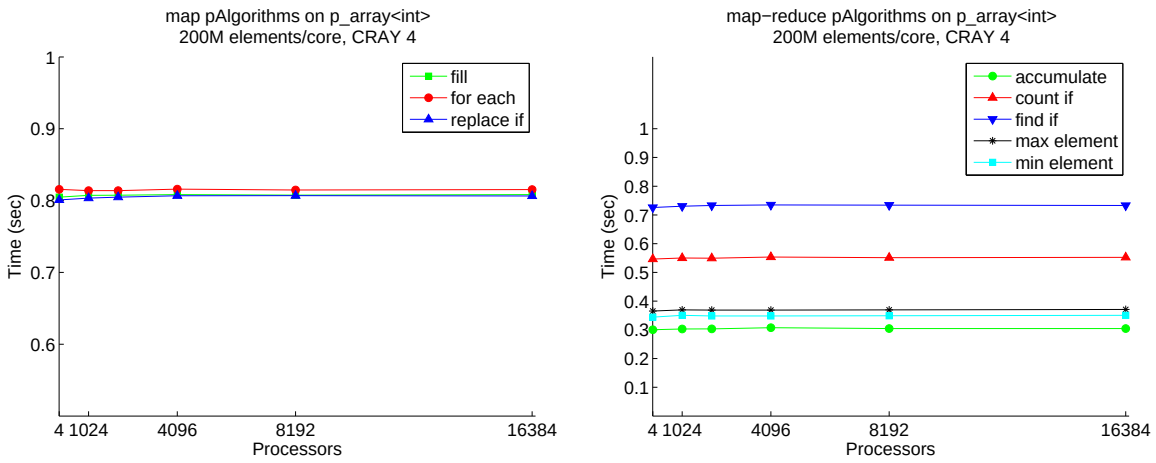
We do, however, begin to see overhead in *localdeps* that is proportional to the number of incremental calls, ranging from 4% to 9%. The *remotedeps* curve is similar to *localdeps*, with some additional time spent in communication to setup and propagate cross processor edges in the task graph. In fact in this case, we can succinctly summarize the utility of incremental generation: Moving from one incremental call to 100 incurs only 5% additional runtime overhead, while allowing us to reduce the `PARAGRAPH` task memory footprint to only 1% of the size of the actual task graph.

VII.3. Generic Algorithms

In this section we show the performance of STAPL on various generic algorithms, first looking algorithms from our library that draw from those defined by STL. We then look at substring matching, illustrating sample STAPL code as compared to an MPI implementation with equivalent functionality. All experiments in the section are weak scaling studies, meaning the amount of data per processor is kept constant. The results of the string matching experiment presented in this section were originally published in [10].

VII.3.1. STL Equivalent Algorithms

Over half of the algorithms in STL can be implemented using the *map* and *map_reduce* patterns. We present a subset of the algorithms using each pattern, running our experiments on CRAY4. The performance of `pAlgorithms` that use the map `task factory` to generate their task graph is shown in Figure 48(a). There are 200 million



(a) (b)

Fig. 48.: Weak scaling of `pAlgorithms` on CRAY4. Those using map and map-reduce patterns are grouped together in (a) and (b), respectively.

integers on each location stored in a `pArray` for each of the results. Similarly, Figure 48(b) shows the results for a experiment on CRAY4 where the `task factory` used by the `pAlgorithm` is map reduce.

All algorithms scale well as the number of cores is increased from 4 to 16,384. The execution time increases less than 1.5% as the number of cores is increased, with the map-based `pAlgorithms` seeing increases in execution time that are closer to 1%.

VII.3.2. String Matching

String matching is implemented by calling `stapl::count_if(view, pred)` with an appropriate `pView` and predicate. In this case, given a pattern of length M, we create an overlapped `pView` over the text, with a core of length 1, left overlap of size 0 and right overlap of size $M - 1$. This will give a `pView` over all the sub-strings

```
struct strmatch {
  const string& S;
  strmatch(const string& s): S(s) {}

  template<typename View>
  bool operator()(View v) const {
    return equal(S.begin(),S.end(),
                 v.begin());
  }
};

void stapl_main(int argc, char** argv)
{
  typedef stapl::p_array<char>
      p_string_type;
  typedef stapl::array_1D_view
      <p_string_type> pstringView;
  ...
  result=stapl::count_if(
           stapl::overlap_view(text,
           1,0,pattern.size()-1),
           strmatch(pattern));
  ...
}
```

(a)

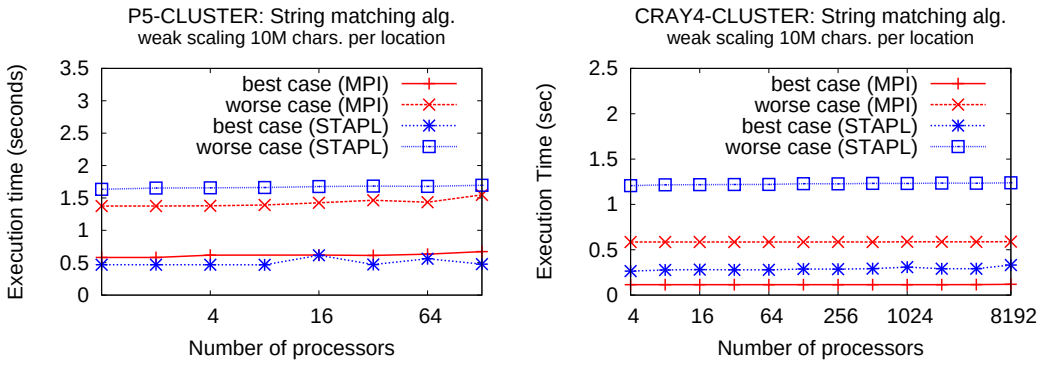
```
int main(int argc, char** argv) {
  ...
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  N=N/P;
  std::vector<char> V(N);
  int M=S.length();

  for (int i=0; i <= N-M+1; ++i)
    if (equal(S.begin(), S.end(),
              V.begin()+i)) ++cnt;
  if (pid>0)
    MPI_Send((&V[0]), M-1, MPI_CHAR,
          pid-1, 1, MPI_COMM_WORLD);
  if (pid<P-1) {
    vector<char> BUFF(2*(M-1));
    copy(V.begin()+N-M+1, V.end(),
              BUFF.begin());
    MPI_Recv( &BUFF[M-1], M-1, MPI_CHAR,
              pid+1, 1, MPI_COMM_WORLD,
              &status );
    for (int i=0; i <= M-1; ++i)
      if (equal(S.begin(), S.end(),
            BUFF.begin()+i )) ++cnt;
  }
  int res;
  MPI_Reduce ( &cnt, &res, 1, MPI_INT,
               MPI_SUM, 0, MPI_COMM_WORLD );
  ...
}
```

(b)

Fig. 49.: String matching algorithm implemented in STAPL and MPI. (a) STAPL code using an overlap partitioned view. (b) MPI version.

of size M of the input text. The code sample is shown in Figure 49(a). In Figure 49(b), an MPI version of the program is shown. In this case it becomes possible to appreciate the additional complexity of the MPI code with respect to the STAPL version, since in MPI the programmer must take explicit care of the boundary regions (this is a special case of the use of ghost nodes, a well known technique in parallel processing [29,40]). Figure 50 shows results for MPI and STAPL on P5-CLUSTER and CRAY4. In both architectures, the performance of the two versions is comparable. In the best case the substring to search is not part of the text, thus the number of



(a) (b)

Fig. 50.: String matching weak scaling on (a) P5-CLUSTER and (b) CRAY4.

occurrences is zero. In the worse case, both text and substring are composed of the same character, maximizing the number of occurrences.

VII.4. NAS Parallel Benchmarks

The NAS Parallel Benchmarks are a set of programs derived from computational fluid dynamics (CFD) applications and contain application kernels that exhibit different computation and communication patterns. Importantly, NAS provides an implementation of the benchmarks, MPI applications written in Fortran and C. This allows us to compare the performance of STAPL against a common implementation. In these results, we use NPB-MPI version 3.3. Various problems sizes (referred to as classes) are specified ranging from small problems for development to problems large enough to run on modern massively parallel systems. Since the problems sizes are fixed by the benchmark specification, these experiments are a strong scaling study.

VII.4.1. NAS EP

EP from the NAS Parallel Benchmarks (NPB) begins by generating n pairs of random numbers (using a deterministic algorithm specified in the standard). For each pair, a small calculation is run to determine if it is a Gaussian pair. If they are, each number in the pair is included in a pairwise global summation. Also accumulated is data about which of 10 annulus rings each pair falls into. Global reductions of two scalars and an array of annulus counts are included in the timed section, and known reference values of the counts are used to validate the run.

The NAS provided MPI implementation is written in Fortran. It computes the local pairs on each processor, and then uses a series of a `mpi_allreduce()` calls to compute the validation set. STAPL employs the *map_reduce* `PARAGRAPH` pattern to specify the computation. Experiments were run on P5-CLUSTER up to 512 processors using gcc 4.5.2 and on CRAY4 up to 16,384 processors using gcc 4.5.1 (gfortran is used for the NAS MPI code). In both cases, 30 iterations were executed on the various processors counts to create the associated 95% confidence intervals for the results. We show graphs for both scalability and execution time.

On P5-CLUSTER, we ran the class B problem ($n = 2^{30}$) with the results shown in Figure 51 and Figure 52. Both applications scale relatively well up to 256, though after this NPB fails show a speedup; and the results become noisy, as can be seen by the large error bars. In contrast, STAPL maintains good scalability up to 512 processors, the largest processor count we can run on this system. We have investigated why NPB performs so poorly at higher processor counts, and it seems to be caused by the three successive `mpi_allreduce()` calls. Removing two of them seems to mitigate the problem, but disallows the kernel from validating. As we will see in the CRAY4 results, NPB exhibits similar behavior at the higher processors counts we tested,

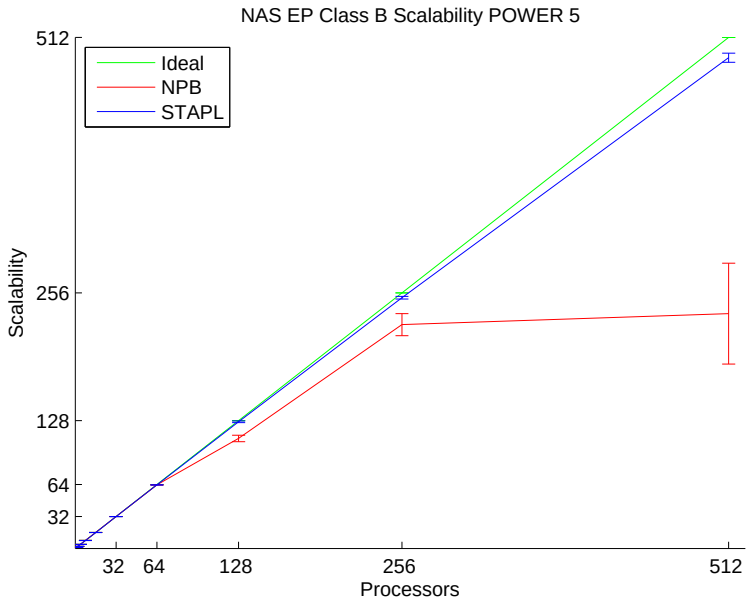


Fig. 51.: NAS EP Class B scalability on P5-CLUSTER.

though the effect is not as pronounced.

The STAPL implementation exhibits a 21% sequential overhead on P5-CLUSTER, and this factor remains constant during the parallel runs. This overhead is not in the initialization but rather in the main computation loop of the kernel. Though we have not discovered the exact cause of this, it appears the different programming languages cause the compiler to generate a slightly different set of instructions. However, due the poor scalability of NPB, we do manage to outperform it in execution time at 512 processors.

We ran both the class C ($n = 2^{32}$) and class D ($n = 2^{36}$) problem sets on CRAY4. The smaller class was run up to 2048 processors, and the results are shown in Figure 53 and Figure 54. For this class, the scalability of both approaches are similar and NPB does not exhibit the problem seen on P5-CLUSTER. Note that STAPL again

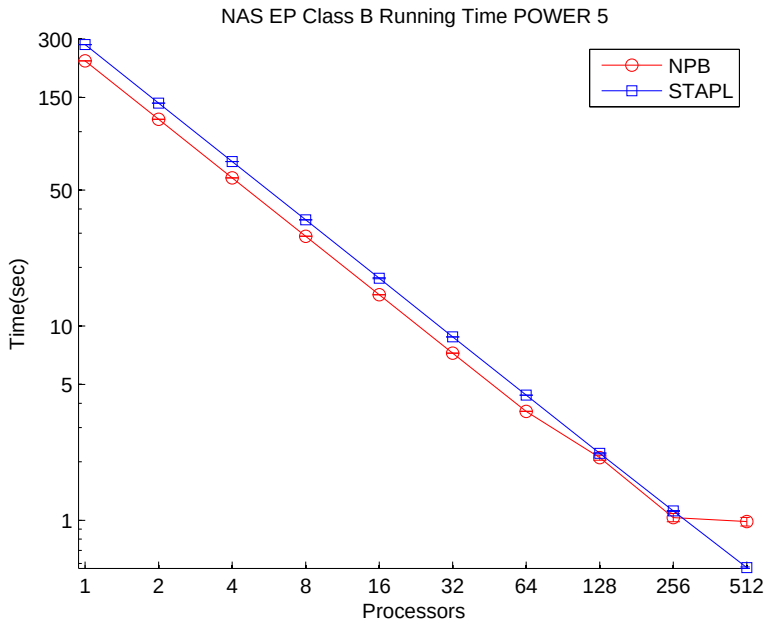


Fig. 52.: NAS EP Class B time on P5-CLUSTER. Logarithmic in both axes.

sees a sequential overhead, this time measuring 47%. The class D results are shown in Figure 55 and Figure 56. STAPL scalability begins drop slightly above at 12K processors. After this point NPB exhibits the same marked drop in performance, again with large variability between the execution iterations. Due to the larger sequential overhead, however, STAPL does not manage to surpass it in performance on the processor counts considered in this experiment.

VII.4.2. NAS CG

The NAS CG benchmark estimates the largest eigenvalue of a symmetric positive definite sparse matrix using the inverse power method. One step of the method requires solving $Az = x$. This computation is accomplished through a sequence of calls to the conjugate gradient method. Both the matrix size and the number iterations of

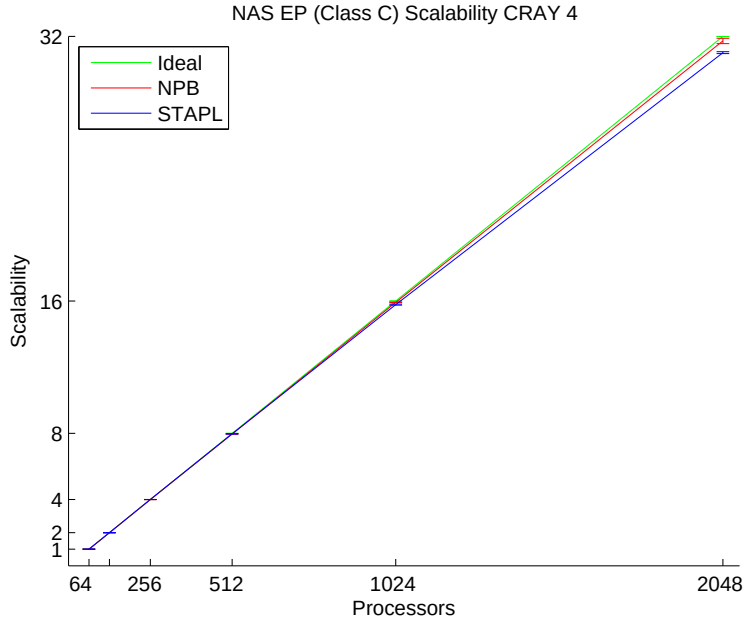


Fig. 53.: NAS EP Class C scalability on CRAY4.

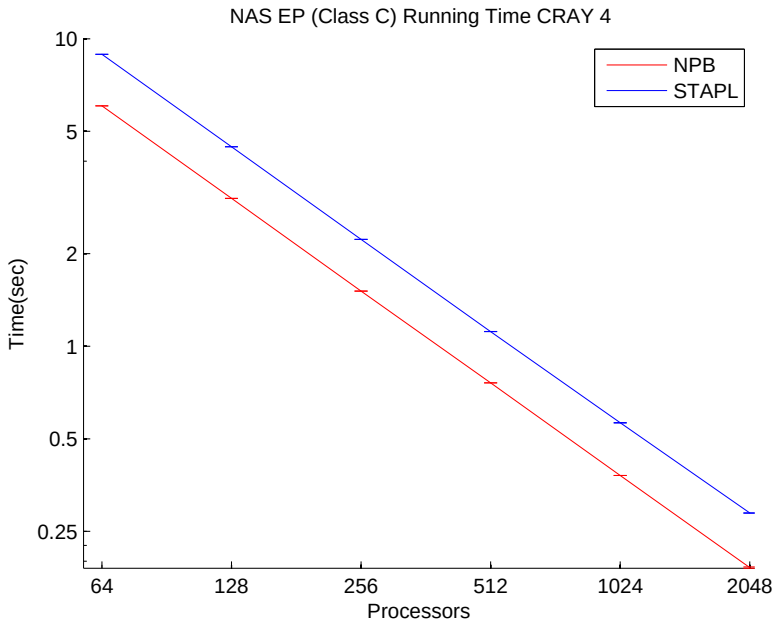


Fig. 54.: NAS EP Class C time on CRAY4. Logarithmic in both axes.

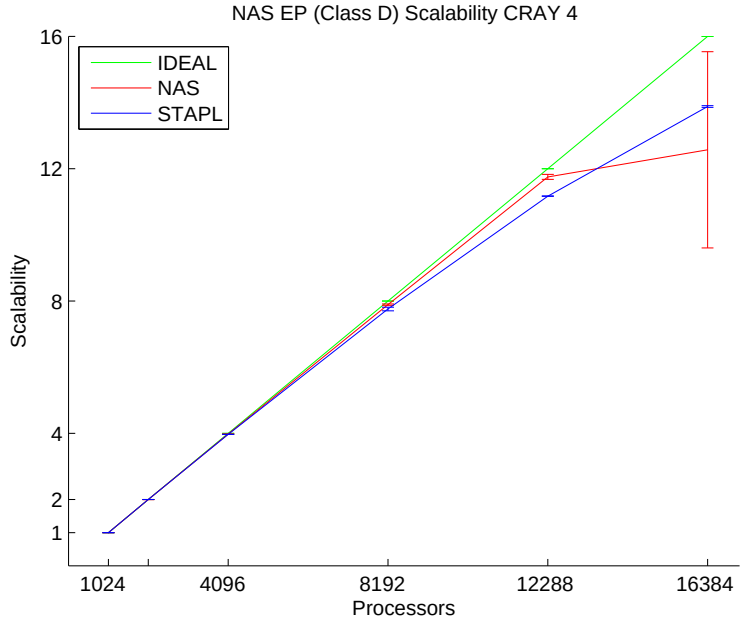


Fig. 55.: NAS EP Class D scalability on CRAY4.

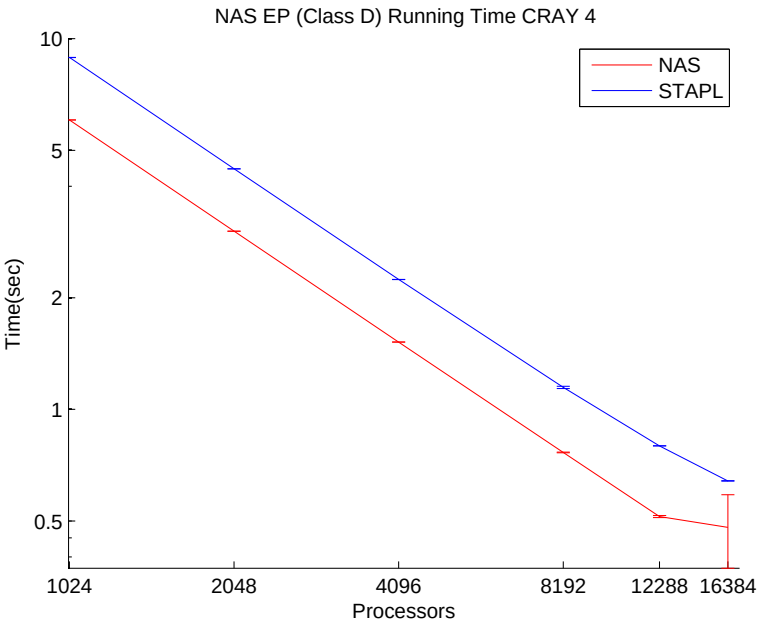


Fig. 56.: NAS EP Class D time on CRAY4. Logarithmic in both axes.

```
1 auto q       = A * p;          // overloaded op calls matvec
2 auto alpha   = rho / inner_product(p, q);
3 auto new_z   = z + alpha * p; // overloaded op calls map_func
4 auto new_r   = r - alpha * q;
5 auto new_rho = inner_product(new_r, new_r);
6 auto beta    = new_rho / rho;
7 auto new_p   = new_r + beta * p;
```

Fig. 57.: STAPL code for a single iteration of the conjugate gradient method.

the power method vary based on the problem class. The conjugate gradient method is fixed at 25 iterations per power method step for all classes. The kernel is typical of unstructured grid computations; and it employs a series of communications involving the vector that test both reduction operations and irregular long distance messages, via a transposition operation in the matrix-vector multiplication which occurs inside every conjugate gradient iteration.

The NAS provided MPI implementation is written in Fortran, and employs the common use of a two dimensional block matrix distribution; this allows each processor to progress while only needing access to $n/\sqrt{p}$ elements of the N size vector, significantly reducing communication bandwidth costs. Written in approximately 1800 lines of code, the implementation uses more sophisticated techniques then EP, employing a series of nonblocking MPI calls to better hide communication latency. In short, it mimics the efforts of a more sophisticated parallel application developer.

The STAPL implementation is written as a composed `PARAGRAPH` using generic algorithms such as the matrix-vector multiplication `PARAGRAPH` discussed in Section III.3. A portion of the STAPL implementation is show in Figure 57. The entire application consists of 550 lines of code, and the `pMatrix` container uses the same 2D

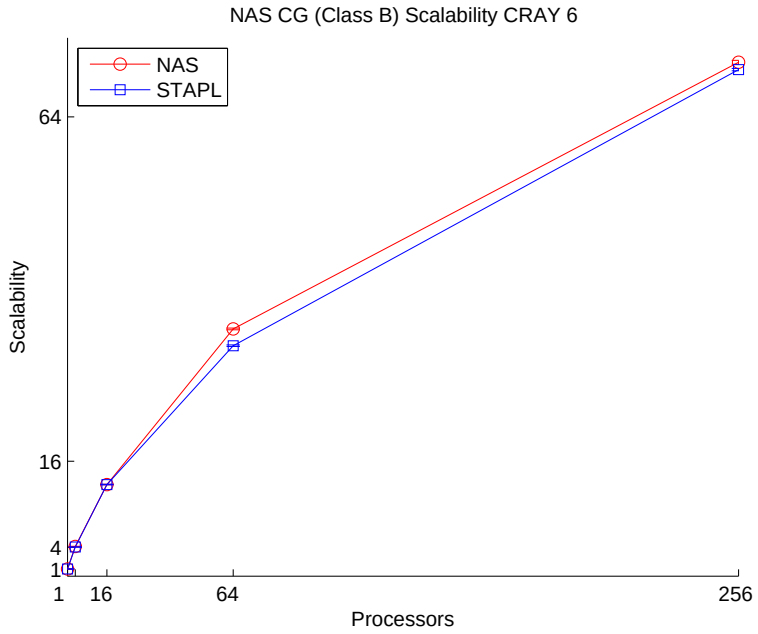


Fig. 58.: NAS CG Class B scalability on CRAY 6.

block matrix distribution as the NAS implementation, enabling the `PARAGRAPH` to employ similar communication optimizations. This kernel demands substantially more of the `PARAGRAPH Executor` than previous experiments, as the top level `PARAGRAPH` creates many nested task graphs during execution. For example, the Class B problem creates a total of 11,000 nested `PARAGRAPHs`, making incremental generation a necessity.

We ran the Class B ($N = 75,000$, $Iter = 75$) and the Class D ($N = 150,000$, $Iter = 100$) problems on CRAY 6. The smaller problem was executed on processor counts up to 256 processor, while the latter began at that count and was studied up to 16,384 processors. For these and all other CG experiments, 30 iterations were executed at each data point to create the associated 95% confidence intervals for the

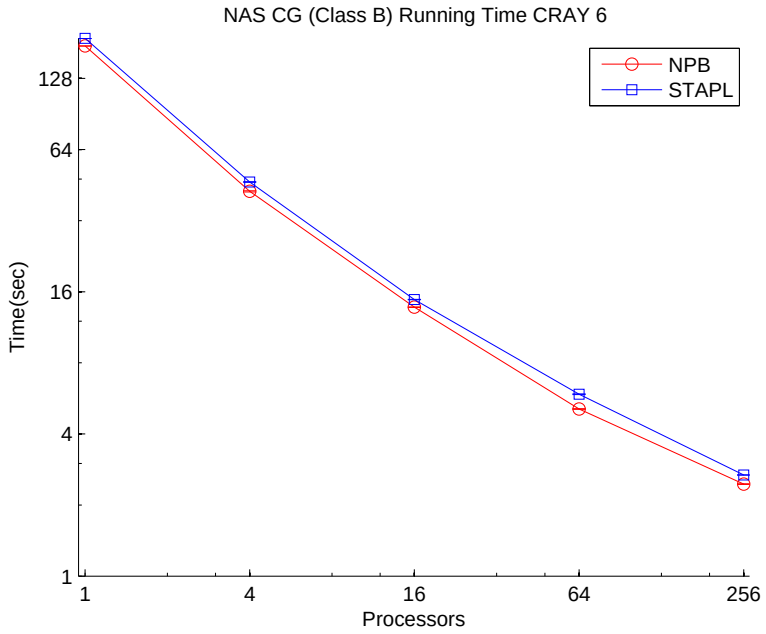


Fig. 59.: NAS CG Class B time on CRAY 6. Logarithmic in both axes.

results. We show graphs for both scalability and execution time.

On CRAY 6, STAPL exhibits 8% sequential overhead relative to the NAS implementation. For the Class B problem (see Figures 58 and 59) this overhead holds for all processor counts, except at 64 processors where this rises to 15% and causes STAPL scalability to suffer slightly. We will discuss possible causes of this behavior when we explain the Class D results. For this problem class, though, the performance of STAPL overall closely mirrors that of the NAS MPI version.

The Class D experiments are shown in Figures 60 and 61. We begin at 256 processors, with STAPL exhibiting similar overhead as in Class B results. However, the trend does not hold, with the two code reaching parity at 1,024 processors and STAPL slightly outperforming NAS at 4,096 processors. This is not completely unexpected, as

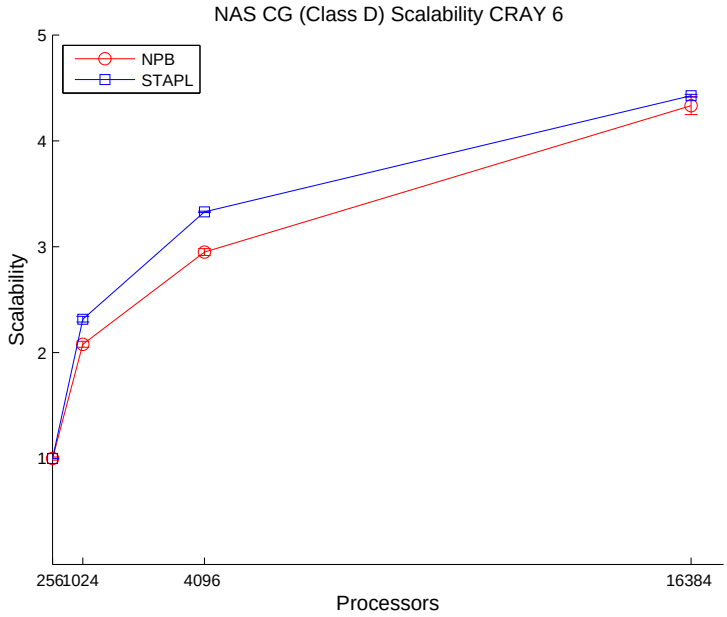


Fig. 60.: NAS CG Class D scalability on CRAY 6.

the `PARAGRAPH` specification should enable the `PARAGRAPH Executor` to better overlap communication and computation than its NAS counterpart. However, the scalability gap closes again at 16,384 processors as STAPL has a relative drop in performance, returning to approximately 8% slower.

We are investigating the source of the performance aberration STAPL sees at the highest processor count on CRAY 6 Class D (and similarly below on P5-CLUSTER). On CRAY 6, the torus interconnect gives each node 6 links for incoming MPI connections: two in the X and Z dimensions and one in the Y dimension. When posting a non-blocking MPI receive request (i.e. `MPI_Irecv()`), which both implementations do, corresponding requests must be posted on all links where that message may arrive. The NAS implementation explicitly computes the source processor from which a destination expects to receive a message. Thus, the message need only be posted

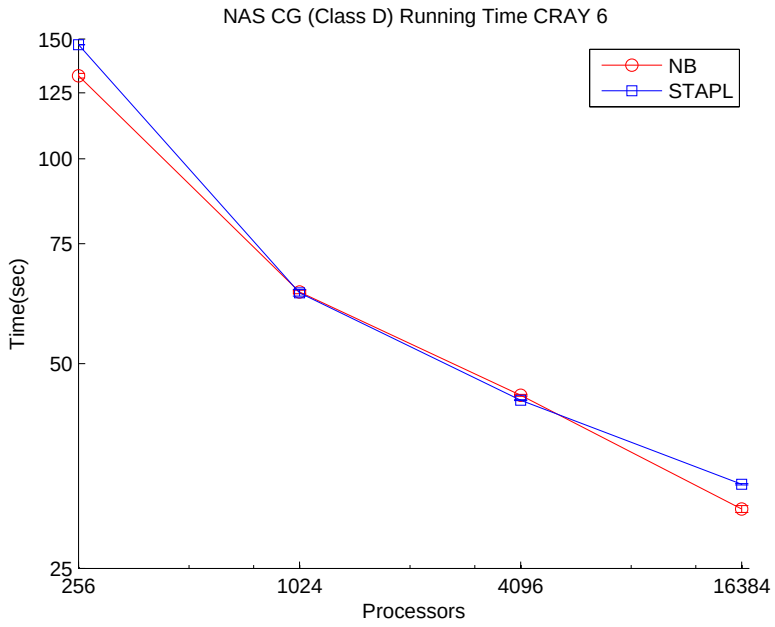


Fig. 61.: NAS CG Class D time on CRAY 6. Logarithmic in both axes.

on one link. `ARMI` in contrast, to support arbitrary communication patterns, posts its receives with the `MPI_ANY_SOURCE` tag. While this allows it to receive messages from unknown senders, it causes inefficiencies, as this request must be posted on all incoming links. Referred to as "promiscuous" receives, they complicate the internal MPI reception protocol as additional care must be taken to properly matches these replicated posted receives with the single receive requested by the MPI user. Initials tests suggest that removing or at least reducing these types of receives will remove at least some of the overhead STAPL sees in these cases. As the `PARAGRAPH Executor` often has this exact communication information, we are working to develop an interface with `ARMI` that allow us express this additional knowledge the runtime system.

We also ran the Class B problem on P5-CLUSTER, and the results are shown in Figures 62 and 63. The behavior is similar to the Class D experiment on CRAY 6,

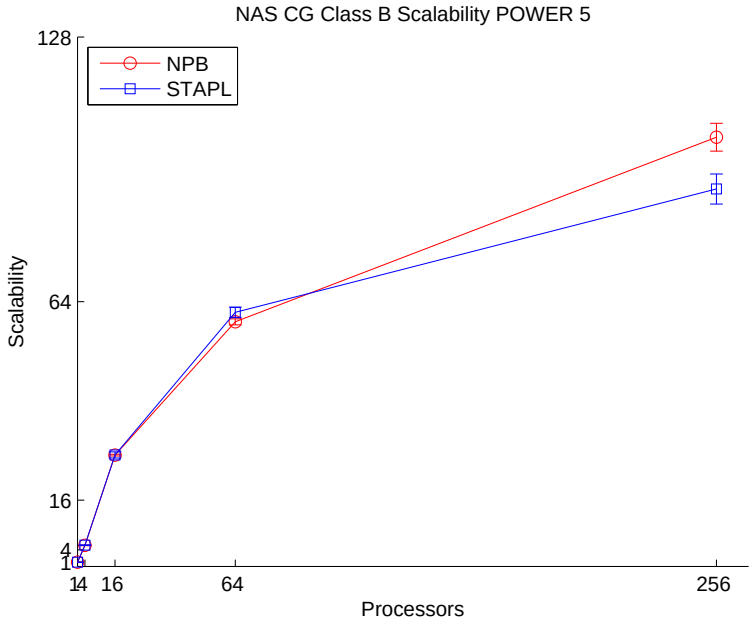


Fig. 62.: NAS CG Class B scalability on P5-CLUSTER.

with STAPL outperforming NAS at the 64 processor level. However, the reversal at the highest processor count is more pronounced here, with a running time difference of 28%, as opposed to 11% in the sequential case.

VII.5. PDT - Discrete-Ordinates Particle Transport

In this section, we describe an application written in STAPL from the nuclear engineering domain and show performance results on CRAY4. PDT solves the discrete-ordinates particle transport problem [2] which can be briefly described as follows. Given:

1. A domain in an N-dimensional space made of known materials,

2. An initial flow of particles through the domain at a starting time,

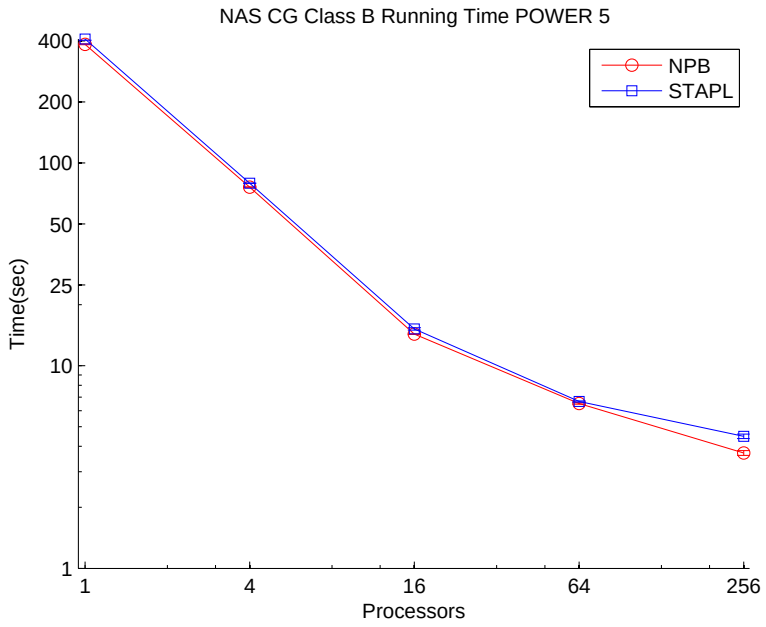


Fig. 63.: NAS CG Class B time on P5-CLUSTER. Logarithmic in both axes.

3. An initial set of sources generating particles inside the domain, and

4. Knowledge about the behavior at the domain boundary,

Compute: the flow of particles at a later point in time for every point in the spatial domain.

In PDT, the discretized spatial domain is represented by a `pGraph`. A series of *sweep* `PARAGRAPHs` are generated from starting points in this domain. In Figure 64, we show a two dimensional example of the `pGraph` representation of the spacial domain, and in Figure 65 we depict the associated task graphs.

We have designed an artificial input for PDT that allows us to perform a weak scaling study. It employs a three dimensional spatial domain which is partitioned among the processors. The `PARAGRAPH` defines a custom `task factory` for the sweep

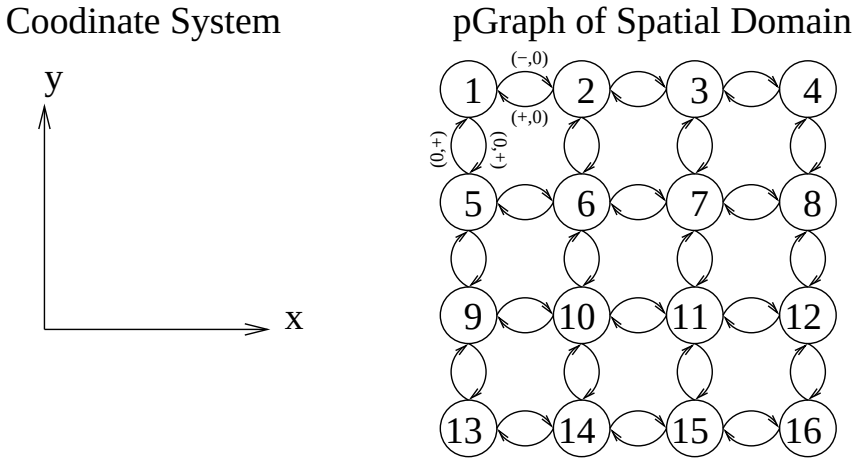


Fig. 64.: Coordinate system and `pGraph` representation of the PDT spatial domain.

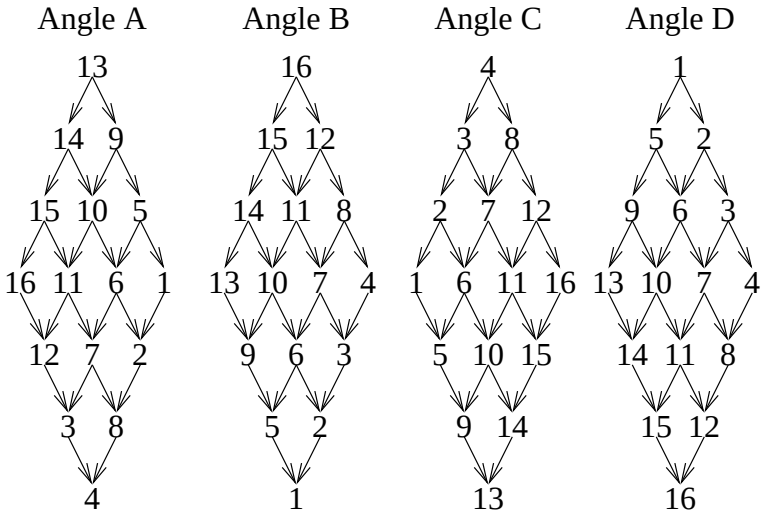


Fig. 65.: Task graphs generated for the set of directions and spatial domain.

and eight instances of these in a manner specifying that they can be run concurrently. We focus on the performance of this composed sweep operation as it is the majority of the execution time in other applications solving the same problems.

Due to the nature of the problem, exact weak scaling behavior (i.e, execution time at varying processor counts remains constant) is not expected, as communication fundamentally increases with the growth in processors. Hence it is expected that execution time of an experiment in this study must increase as the number of processors increases. Our collaborators model this behavior, determining an expected running time based on the processor count, problem input characteristics, and estimations of the communication costs on the target architecture. This model estimates a lower bound on execution time for our experiments, and does not include estimates for the overhead of the `PARAGRAPH Executor`.

Figure 66 shows the execution time of PDT on CRAY4 and the estimated execution time produced from the model. The execution times we observe follow the same trend predicted by the model with overhead at most of 9% occurring at 2,048 processors. The overhead does increase slightly as the processor count increases. This is somewhat expected in the current experimental setup, as the method to scale the problem to larger processor counts causes the number of tasks that the `PARAGRAPH` generates on each processor to be greater (while the work per task decreases). Therefore the amount of bookkeeping the `PARAGRAPH Executor` must do is greater. We are working with our collaborators to either improve the model to account for this task increase or alternatively to modify the experimental setup to remove the need for these additional tasks.

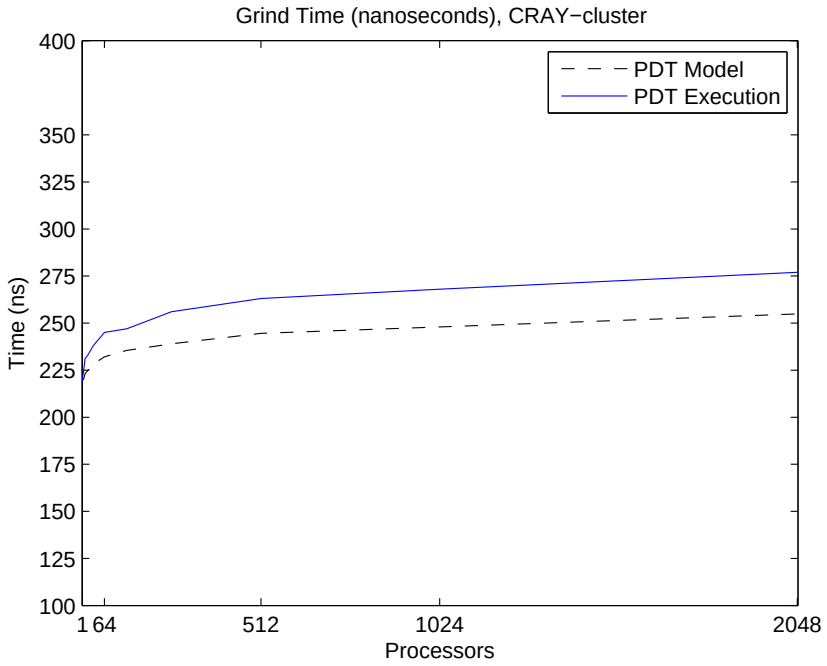


Fig. 66.: Weak scaling of PDT on CRAY4.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

Parallel programming has become increasingly important in mainstream computing, as multiprocessor and multicore architectures continue to increase in both availability and processing element counts. Furthermore, in high performance computing, the need to solve larger and more complex problems has pushed the definition of massively parallel systems to higher and higher levels, a trend that appears will continue for some time. To effectively harness the capabilities of these new architectures, new programming paradigms are needed that enable users to productively develop efficient parallel applications.

STAPL is being developed with this goal in mind. Once of its key design decisions is to utilize task graph composition as the programming model to construct new algorithms and applications. The `PARAGRAPH` infrastructure allows developers to focus on expressing the fundamental parallelism and data dependences in the applications, without worrying about lower level issues such as data locality, work distribution, and dependence enforcement. In this dissertation, we have presented the mechanisms the `PARAGRAPH Executor` employs to map these task graph specifications to a target architecture in a efficient and scalable manner.

We began by providing some new insight into the dependence specification problem, presenting an approach that removes non-fundamental synchronizations, allowing task graph initialization and execution to occur simultaneously. Related to this relaxed ordering is support for incremental graph generation, removing the need to have the fully specified task graph ever exist at once in its entirety. We then describe a taxonomy of workfunctions that can be used to populate this hierarchical task graph, providing a unified model for task and data parallelism.

In the implementation description, we showed how our event driven architecture allows both task creation and task execution to drive changes in the edge container, the primary data structure responsible for dependence enforcement, value propagation, and task retirement. These activities employ the task identifier directory, which allows the edge container to operate in an environment where tasks are dynamically positioned in the system based on runtime conditions. Finally, the runtime scheduler manages the set of active task graphs, scheduling tasks for execution and determining when a graph has finished execution so that its remaining allocated resources can be reclaimed.

We next discussed two important `PARAGRAPH Executor` optimizations that attempt to minimize data access costs. Task placement combats latency by migrating computation near the data it operates on, employing a customizable policy to guide this activity. Second, view localization removes the overhead associated with distributed memory address translation, detecting when more direct access to elements can be safely utilized. Then, in the evaluation chapter, we explore the performance of our task graph infrastructure. We demonstrate that STAPL algorithms and applications specified using `PARAGRAPHs` to create high level task graphs can be efficiently mapped onto modern parallel systems. The combination of the approaches presented in the dissertation enable scalable performance at a high degree of parallelism.

There are several promising avenues for future work in the `PARAGRAPH Executor`. First, it would be interesting to implement some of the other edge specification approaches discussed in Section IV.2.3. Both the relative performance of the associated edge container implementations and the change in the expression of task graphs by workfunctions merit further investigation.

As suggested earlier, modifying the edge container to operate in a hierarchical manner should not prove difficult and presents some interesting opportunities. Cor-

responding changes would need to be made to the identifier directory. Together, they should enable us to better adapt to the various levels of the memory hierarchy and communication subsystems on a target machine. Neighborhoods of locations could aggregate requests, reducing bandwidth utilization in the system. Furthermore, grouping requests from multiple locations in this way may allow us to determine some structure in the task graph and optimize the way notifications are disseminated.

However, perhaps most importantly, future work should aim to increase the flow of information from the `PARAGRAPH` specification to the `PARAGRAPH Executor` and the STAPL runtime system. We have already identified some traits such as view access modifiers that aid us in our optimization efforts. It is important to identify additional contextual description that is reasonable and appropriate for developers to provide that can inform implementation specializations at the lower level. Appropriately defining this interaction between application specification and execution will not only increase performance, but further promote a separation of concerns needed for higher productivity in the development of parallel programs.

REFERENCES

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Boston: Addison-Wesley Professional, 2004.

[2] M.L. Adams and E.W. Larsen, "Fast iterative methods for discrete-ordinates particle transport calculations," *Progress in Nuclear Energy*, vol. 40, no. 1, pp. 3–159, 2002.

[3] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol 29, no. 12, pp. 66–76, 1996.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger, "Stapl: An adaptive, generic parallel c++ library," in *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, KY, Aug. 2001, pp. 195–210.

[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger, "Stapl: A standard template adaptive parallel c++ library," in *Proceedings of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul. 2001.

[6] M. Austern, "Draft technical report on c++ library extensions," ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Tech. Rep. n1836=05-0096, 2005.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakr-

ishnan and S. Weeratunga, "The NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep. NAS RNR-94-007, 1994 [Online]. Available: http://www.nas.nasa.gov/npb/

[8] H. Bischof, S. Gorlatch, and R. Leshchinskiy, "Generic parallel programming using c++ templates and skeletons," in *Domain-Specific Program Generation,* C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, Eds., London: Springer, 2003, pp. 107–126.

[9] G. Blelloch, "NESL: A Nested Data-Parallel Language," Carnegie Mellon University, Tech. Rep. CMU-CS-93-129, 1993.

[10] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl pview," in *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Houston, TX, Sep. 2010, pp. 261–275.

[11] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "Stapl: Standard template adaptive parallel library," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR)*, Haifa, Israel, May 2010, pp. 1–10.

[12] A. Buss, T. Smith, G. Tanase, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "Design for interoperability in stapl: pMatrices and linear algebra algorithms," in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Edmonton, Alberta, Canada, Jul. 2008, pp. 304–315.

[13] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Los Alamitos, CA, 2004, pp. 52–60.

[14] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Atlanta, GA, Apr. 2010, pp. 1–12.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, San Diego, CA, Oct. 2005, pp. 519–538.

[16] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and implementation of the scalapack lu, qr, and cholesky factorization routines," *Scientific Programming*, vol. 5, no. 3, pp. 173–184, 1996.

[17] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge: MIT Press, 1991.

[18] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd edition. Cambridge: MIT Press, 2001.

[20] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von

Eicken, and K. Yelick, "Parallel programming in split-c," in *Proceedings of the 7th ACM International Conference on Supercomputing (ICS)*, Tokyo, Japan, Nov. 1993, pp. 262–273.

[21] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Mar. 2004, pp. 137–150.

[22] E. Deelman, G. Singh, M. Su, J. Blythe, A. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

[23] J. Demmel, J. Dongarra, B. N. Parlett, W. Kahan, M. Gu, D. Bindel, Y. Hida, X. S. Li, O. Marques, E. J. Riedy, C. Vömel, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Langou, and S. Tomov, "Prospectus for the next lapack and scalapack libraries," in *Proceedings of the 8th Workshop on Parallel Scientific Computing (PARA)*, Umeå, Sweden, Jun. 2006, pp. 11–23.

[24] I. T. Foster, J. S. Vöckler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," in *Proceedings of the 14th International Conference on Scientific and Statistical Database Management (SSDBM)*, Edinburgh, Scotland, Jul. 2002, pp. 37–46.

[25] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*. Reading: Addison-Wesley, 1999.

[26] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the 19th ACM SIGPLAN Conference on*

Programming Language Design and Implementation (PLDI), Montreal, Quebec, Canada, 1998, pp. 212–223.

[27] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[28] M. Girkar and C. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[29] J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua, "Writing productive stencil codes with overlapped tiling," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 1, pp. 25–39, Jan. 2009.

[30] J. L. Hennesy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Edition. San Francisco: Morgan Kaufmann, 1996.

[31] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989.

[32] *Reference Manual for Intel Threading Building Blocks, version 1.13*, Intel Corporation, Santa Clara, CA, 2009.

[33] *Reference Manual for Intel Threading Building Blocks, version 1.24*, Intel Corporation, Santa Clara, CA, 2009.

[34] E. Johnson, "Support for Parallel Generic Programming," PhD thesis. Indiana University, Indianapolis, IN, 1998.

[35] J. Kim, M. Spraragen, and Y. Gil, "An intelligent assistant for interactive workflow composition," in *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI)*, Lisbon, Portugal, Feb. 2004, pp. 125–131.

[36] K. Knobe, "Ease of use with concurrent collections," in *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar)*, Berkeley, CA, Mar. 2009, pg. 17.

[37] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," in *Proceedings of ACM 2001 Java Grande / ISCOPE Conference*, Stanford, CA, Jun. 2001, pp. 21–29.

[38] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[39] T. J. Lehman, S. W. Mclaughry, and P. Wycko, "Tspaces: the next wave", in *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS)*, Maui, HI, Jan. 1999, pg. 9.

[40] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[41] H. A. Mandviwala, U. Ramachandran, and K. Knobe, "Capsules: Expressing composable computations in a parallel programming model," in *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana-Champaign, IL, Oct. 2007, pp. 276–291.

[42] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide*, 2nd. edition. Reading: Addison-Wesley, 2001.

[43] National Energy Research Scientific Computing Center, *Franklin: NERSC's*

CRAY XT4 System, Jun. 2011, http://www.nersc.gov/users/computational-systems/franklin/.

[44] National Energy Research Scientific Computing Center, *Hopper: NERSC's CRAY XE6 System*, Jun. 2011, http://www.nersc.gov/users/computational-systems/hopper/.

[45] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors," *International Journal of High Speed Computing*, vol. 1, no. 1, pp. 45–72, 1989.

[46] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library," in *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998, pp. 402–409.

[47] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn, "POOMA: A framework for scientific simulations of parallel architectures," in *Parallel Programming in C++*, G. V. Wilson and P. Lu, Eds., Cambridge: MIT Press, 1996, pp. 547–588.

[48] S. Saunders and L. Rauchwerger, "ARMI: An adaptive, platform independent communication library," in *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, Oct. 2003, pp. 230–241.

[49] A. B. Sinha, L. V. Kale, and B. Ramkumar, "A dynamic and adaptive quiescence detection algorithm," Parallel Programming Laboratory, Department of

Computer Science, University of Illinois Urbana-Champaign, Tech. Rep. 93-11, 1993.

[50] T. G. Smith, "Parallel program composition with paragraphs in stapl," PhD thesis. Texas A&M University, College Station, TX, 2012.

[51] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd edition. Cambridge: MIT Press, 1998.

[52] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulus, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl parallel container framework," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Antonio, TX, Feb. 2011, pp. 235–246.

[53] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger, "Associative parallel containers in stapl," in *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana-Champaign, IL, Oct. 2007, pp. 156–171.

[54] G. Tanase, X. Xu, A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl pList," in *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Wilmington, DE, Oct. 2009, pp. 16–30.

[55] I. G. Tanase, "The stapl parallel container framework," PhD thesis. Texas A&M University, College Station, TX, 2010.

[56] Texas A&M Supercomputing Facility, Texas A&M University, *Hydra: An IBM p5-575 Cluster*, Jun. 2011, http://sc.tamu.edu/systems/#hydra.

[57] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger, "ARMI: A high level communication library for stapl," *Parallel Processing Letters*, vol. 16, no. 2, pp. 261–280, 2006.

[58] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in stapl," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Chicago, IL, Jun. 2005, pp. 277–288.

[59] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11, pp. 825–836, 1998.

VITA

Nathan Lee Thomas received his B.S. in computer science from Texas A&M University in May 1999. He graduated Summa Cum Laude. During his graduate career, he received a Department of Education Graduate Assistance in Areas of National Need (GAANN) Fellowship.

His research focus is parallel programming tools and their role in simplifying the development of applications deployed on massively parallel systems. He is also interested in programming languages and machine learning. He received his Ph.D. in Computer Science from Texas A&M University in May 2012.

More information about his research and publications may be found at http://parasol.tamu.edu/people/~nthomas. He may be reached at: Parasol Lab, 301 Harvey R. Bright Bldg, 3112 TAMU, College Station, TX 77843-3112. His email address is nthomas@cse.tamu.edu.

The typist for this dissertation was Nathan Lee Thomas.